

Arquitectura de computadores

Autor: Pau Arlandis Martínez

Sobre las normas de la asignatura

Teoría

4 Prácticas (voluntarias). Solo puedes presentarte una vez.

- Entrada/Salida → Simulador 15%
- Memoria caché
- Pipeline de instrucciones } MC 88110 35%
- Multiprocesadores → C 25%

Las prácticas pueden sumar, juntas, hasta un punto más en la nota de teoría.

3 parciales

- Tema 1 → 30%
- Tema 2 → 40%
- Tema 3 y 4 → 40% (Posibilidad de recuperar tema 1 [25%] o tema 2 [35%])

Debe obtenerse una nota mínima de 2 puntos en cada uno para que hagan media.

Práctica

1 proyecto (hasta la semana 9)

- Entrada/ Salida.
- Examen (al terminar, semana 9).
- Memoria escrita.

Nota final

Siempre que ambas partes (teoría y proyecto) estén aprobadas (con más de un 5 cada una) la nota final se calcula mediante la fórmula:

$$0.7 \times nota_{teoría} + 0.3 \times nota_{proyecto}$$

Tema 1 – Entrada / Salida

Profesor: Manuel Nieto

Introducción

Periféricos

Existen multitud de periféricos, con diferentes características cada uno:

- Modo de funcionamiento.
- Formato y tamaño de los datos. (Cuántos se intercambian, si este tamaño está definido...)
- Velocidad.
- Tiempo de acceso.

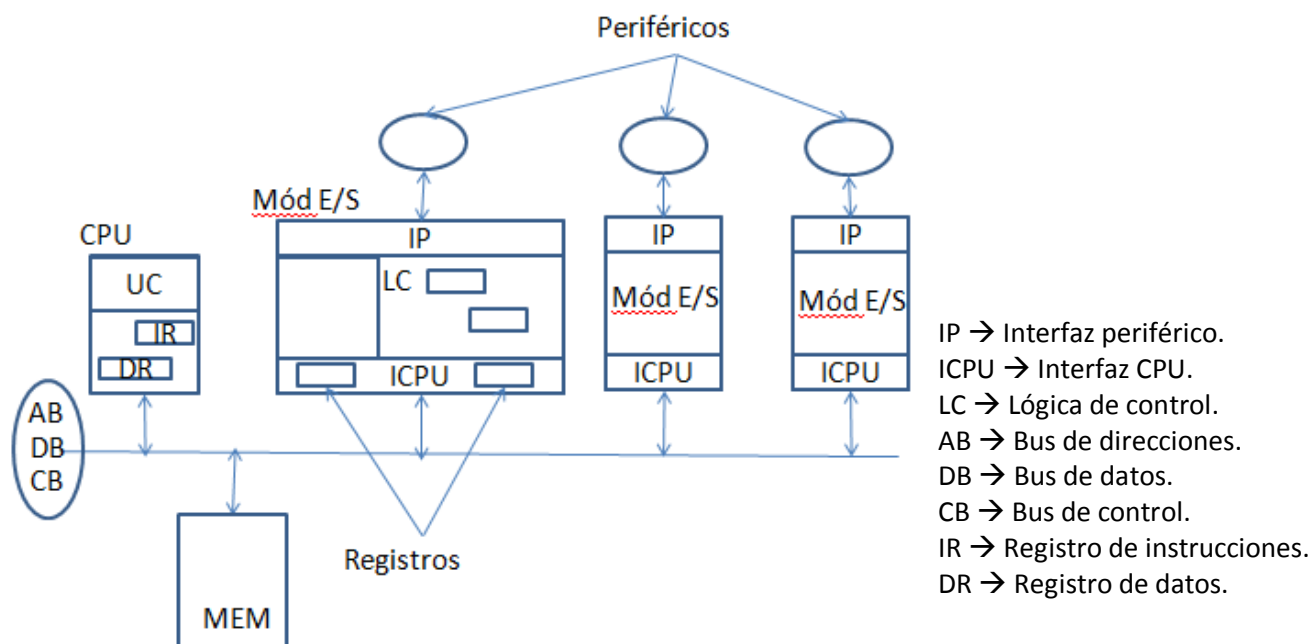
Módulo de E/S

Se hace necesaria **unificar** la visión hardware de los periféricos. Para ello nace el **módulo de E/S** que oculta las particularidades de cada periférico. La CPU solo dialoga con los módulos, que son todos iguales, estándar. Por tanto, tiene dos interfaces: Una que se comunica con el periférico y otra con el CPU.

Tiene varias funciones:

- Control...
- ...
- Comunicación con el periférico → siguiendo las características del mismo.
- Buffering. Dado que cada periférico tiene una velocidad diferente el módulo tiene capacidad de almacenamiento para poder controlar dicha velocidad y adaptarla a la de la CPU.
- Control de errores y situaciones anómalas.

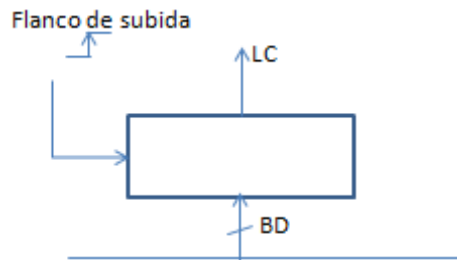
La estructura de un módulo de E/S sería algo similar a:



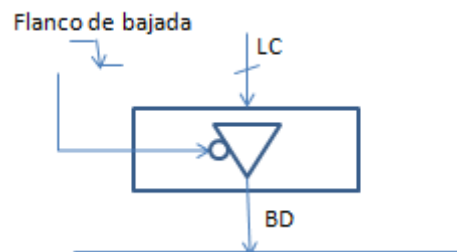
Lo único que controlamos son los registros de la Interfaz con la CPU. Estos son de tres tipos:

Registros de datos

- **Registros de salida.** Un conjunto de biestables que nos permiten sacar información desde el bus de datos a la lógica de control.



- **Registro de entrada.** Saca información de la LC y la vuelca en el bus de datos. La salida se controla mediante un buffer triestado con estados 0, 1 y nada, vacío.



Registros de estado

Son los que indican el estado del periférico y permiten el control de errores.

Registros de control

Se encargan del control de todos los componentes y registros del módulo.

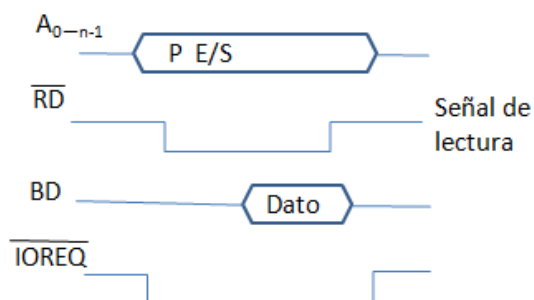
Instrucciones de los módulos

Existen dos formas de dirigir el control de los módulos y de mapear sus direcciones.

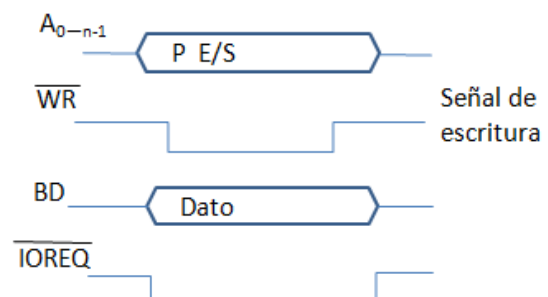
Mapas separados

Para controlar los módulos E/S con este método es necesario establecer un par de ciclos nuevas que permitan la entrada y la salida desde el procesador a los módulos.

Ciclo de entrada

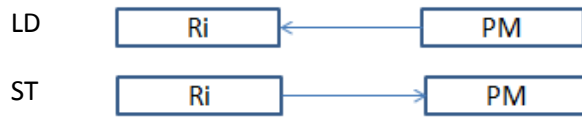


Ciclo de salida

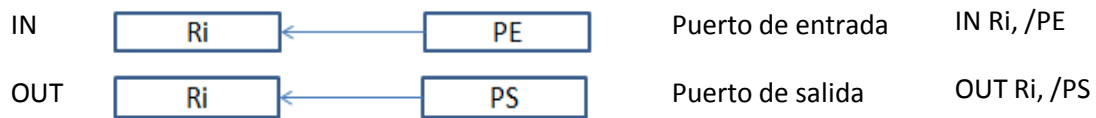


Para que la memoria codifique, lea o modifique la información del bus se activa la señal $\overline{\text{MEMRQ}}$, para los módulos de E/S haremos igual. Crearemos la señal $\overline{\text{IOREQ}}$, si no está activa no hace nada la CPU, pero si lo está inicia la ejecución en el módulo.

A nivel de instrucciones sucede lo mismo, existen dos instrucciones para acceder a la memoria existen dos instrucciones:

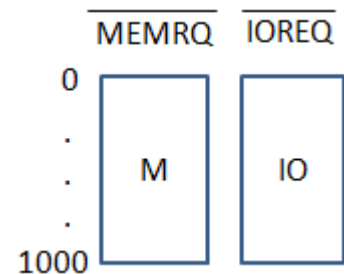


De forma paralela trabajaremos con los módulos, en los que existirán dos instrucciones:



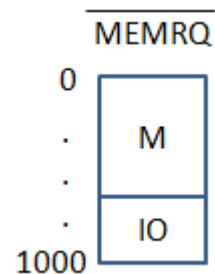
En este caso tendremos dos rangos de direcciones diferentes:

- 0-1000(por ejemplo) con $\overline{\text{MEMRQ}}$ activada.
- 0-1000 con $\overline{\text{IOREQ}}$ activada.

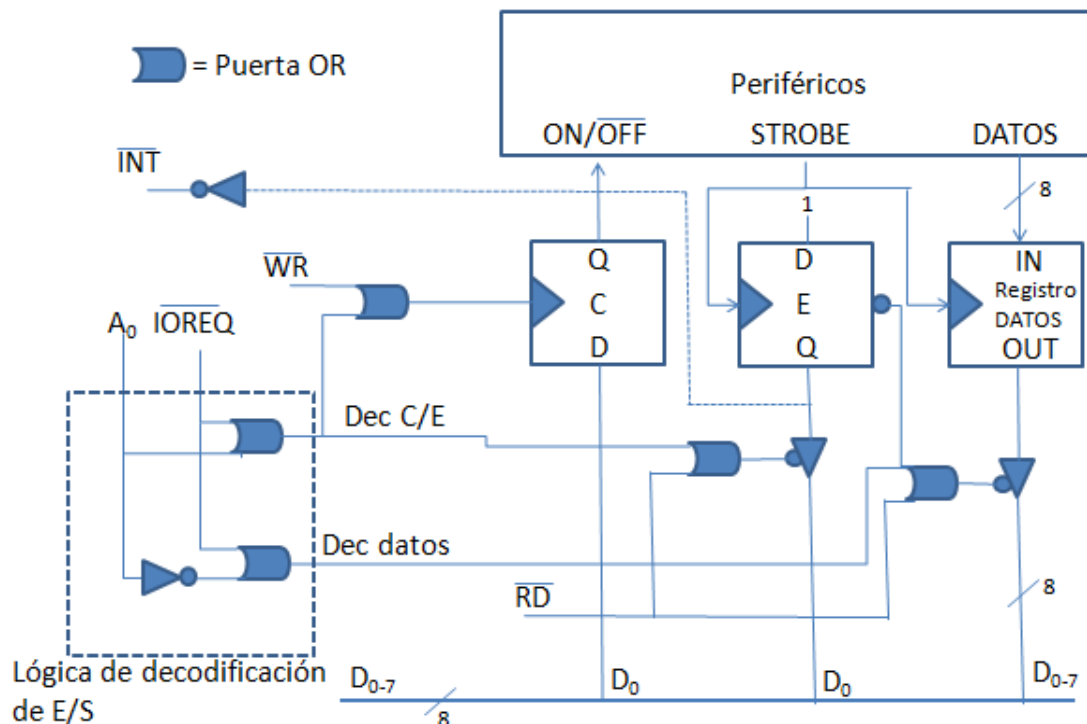


Mapeado en memoria

En este caso solo existe un rango de direcciones ya que no se dispone de señal $\overline{\text{IOREQ}}$, entonces no existen las instrucciones IN y OUT, solo las de direccionamiento de memoria. La solución que se toma es reservar un subrango de direcciones de la memoria para las de E/S. Debe tomarse un rango al principio o al final para mejorar la eficiencia.



Ejemplo de módulo de E/S



Vamos a suponer que todas las direcciones son para este periférico ya que es único. Las direcciones que tengan un 0 en el bit menos significativo (A_0) hacen referencia al registro de control (C) o al de estado (E). Si tienen un 1, hacen referencia al de datos (D).

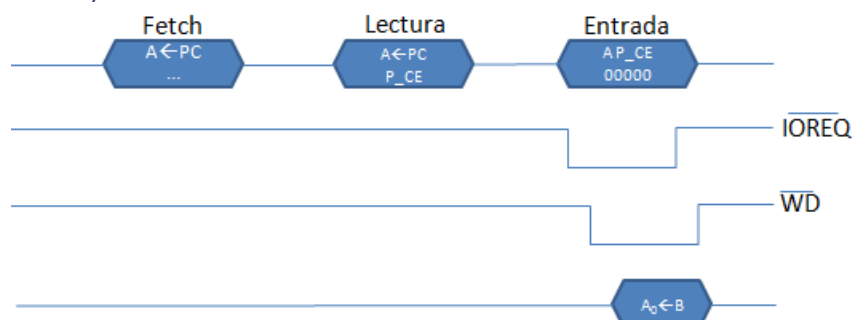
- $xx...x0 \rightarrow P_CE$, por ejemplo $\rightarrow 000...0$
- $xx...x1 \rightarrow P_D$, por ejemplo $\rightarrow 000...1$

Para cada registro existen, pues, condiciones para su lectura o escritura:

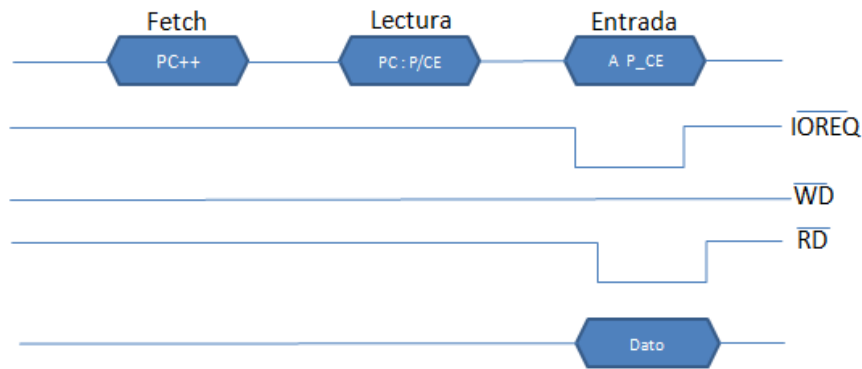
- C. Se activa si tenemos una dirección par ($A_0=0$), si la señal de entrada/salida está activa (IOREQ) y si la señal de escritura (WR) está activa.
- E. Se activa si tenemos una dirección par, si la señal de entrada/salida está activa y si la señal de lectura (RD) está activa.
- D. Se activa si tenemos una dirección impar ($A_0=1$), si la señal de entrada/salida está activa y si la señal de lectura está activa.

Cronograma

in .R0, /P_CE



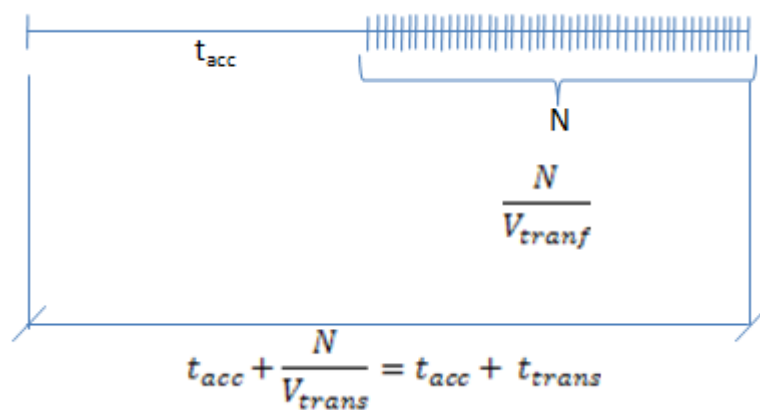
out .R1, /P_CE



En este caso el dato es el registro de estado.

Operación de E/S

Operación de intercambio de un bloque de datos de n bytes entre un periférico y la memoria. El tiempo de acceso es el tiempo que tarda en llegar el primer dato del bloque a la memoria, a partir de entonces comienzan a transmitirse a una determinada velocidad de transferencia.



Técnicas de E/S

Las técnicas de E/S nos permiten saber cómo organizar el intercambio de datos y las instrucciones de entrada/salida para aprovechar mejor el trabajo de la CPU y así ejecutar más programas.

Existen tres técnicas:

- E/S programada.
- E/S por interrupciones.
- E/S por DMA.

Ordenadas por orden de uso de la CPU, de mayor a menor.

Tienen en común 4 fases básicas:

- Iniciar (I). Prepara la operación. Prepara el buffer de E/S.
- Sincronizar (S). Comprueba si el siguiente dato está listo. Si lo está comienza la transmisión.
- Transferir (T). Reg. Datos → M ó M → Reg. Datos. Vuelve a la fase S y espera a que otro dato esté listo.
- Finalizar (F). Dialoga con el periférico para conocer su estado final y qué debe hacer a continuación.

Visto sobre una línea temporal:

I S T S T ... T S T F

Programada

Ejemplo

Datos del procesador

- 100 MIPS (Millones de Instrucciones Por Segundo)
- $t_{acc} = 1ns$
- $V_{transf} = 10^6$ B/seg
- Dir_CE → Dirección de Control y Estado
- Dir_D → Dirección de datos
- ON = 1111 ; 0001
- OFF = 0000
- LISTO = 00001 (X X X X V)

Escribir en ensamblador cada una de las fases básicas de entrada/salida:

```

I { LD .R1, #Dir_Alm_M      ;Dirección de almacenamiento de memoria.
  LD .R2, #1000             ;Contador de datos
  LD .R0, #01
  OUT .R0, /Dir_CE          ;Almacena el contenido de R0(1) en el
                           ;módulo y se conecta

S { SIG: IN .R0, /Dir_CE     ;xxxxV
  AND .R0, #0001            ;0000V
  CMP .R0, #LISTO           ;Listo es V
  BNE $SIG                  ;si no son iguales mantiene el bucle hasta
                           ;que lo sea.

T { IN .R0, /Dir_datos       ;Leemos el dato del periférico
  ST .R0, [.R1++]           ;Guardamos el dato en la memoria.
  DEC .R2                   ;Decrementamos el contador de datos porque
                           ;ya ha llegado uno.
  BNZ $SIG                  ;Si no se han enviado todos los datos
                           ;continúa el bucle.

F { LD .R0, #0000
  OUT .R0, /Dir_CE

```

$$t_{opES} = t_{INI} + t_{acc} + \frac{N}{V_{transf}} + t_{TE} + t_F \cong t_{acc} + \frac{N}{V_{transf}} = 1\text{ ms} + \frac{1000}{10^6} \cong 2\text{ ms}$$

Pero de este tiempo solo hacemos trabajo útil durante el tiempo de transferencia elemental (t_{TE})

$$t_T = 1000 \times 4 \text{ ns} = 1000 \times \frac{4}{10^8} = 40 \mu\text{s} \text{ es el tiempo de transferencia elemental}$$

$\frac{4}{10^8}$ es la inversa de MIPS $\frac{1}{100 \times 10^6}$ por 4 Instrucciones.

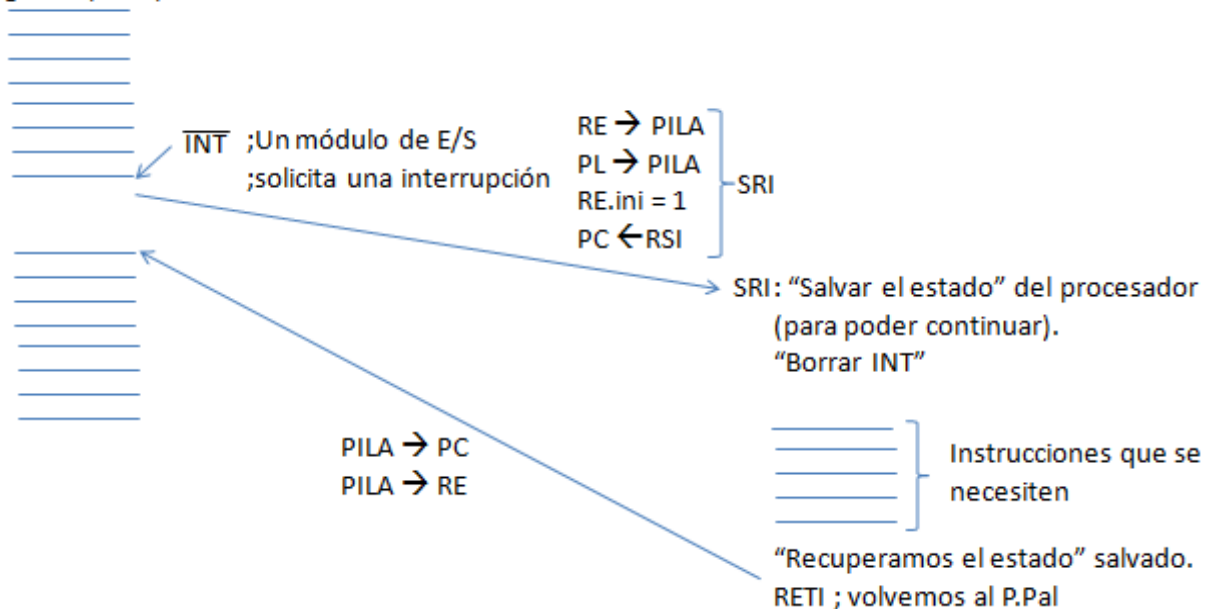
Esto quiere decir:

$$\frac{40}{2000} \times 100 = 2\% \text{ es el porcentaje de tiempo del trabajo útil.}$$

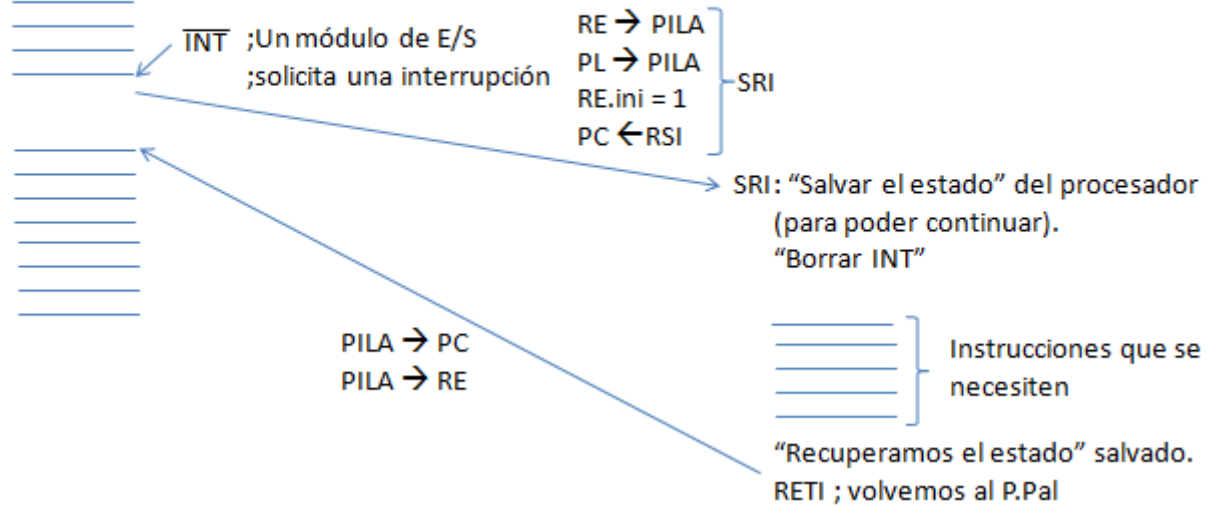
$100 - 2 = 98\%$ es el tiempo que pasamos en el bucle de sincronización.

Interrupciones

Programa principal



Programa principal



Al volver de la interrupción es importante que todos los registros sean idénticos a como estaban antes de la interrupción o podremos cometer un error y que el programa no ejecute correctamente.

Ejemplo

```
LD .R1, #Dir_alm_mem      ; Dirección de almacenamiento de
                          ; memoria.
ST .R1, /Dir_dir_alm_mem  ; Guardamos la dir_alm_mem que
                          ; desconocemos en una variable global
                          ; conocida por todos y que no pueda
                          ; variar otro proceso.

LD .R2, #1000
ST .R2, /Dir_contador    ; Ídem con el contador.
LD .R0, #01              ; "ON".
OUT .R0, /Dir_CE         ; Comienza a funcionar el periférico.
BR $Planificador         ; Cedemos el control al OS.
; Tras las instrucciones que sean que nada tienen que ver con el
; módulo.
RSI: PUSH .R0
    PUSH .R1
    PUSH .R2              ; Almacenamos en la pila el
                          ; estado actual del programa.

    LD .R1, /Dir_dir_alm_mem ; Recuperamos la dirección donde
                          ; podemos almacenar la memoria.

    LD .R2, /Dir_contador   ; Recuperamos el contador.
    IN .R0, /Dir_datos
    ST .R0, [.R1++]
    ST .R1, /Dir_dir_alm_mem
    DEC .R2                ; Decrementamos R2.
    CALLZ $FIN             ; Si R2 es 0, hemos terminado la
                          ; interrupción y salta a la
                          ; rutina que la finaliza.

    ST .R2, /Dir_contador
    POP .R2
    POP .R1
```

```

POP .R0
RETI

FIN: LD .R0, #00          ; "OFF"
     OUT .R0, /Dir_CE
     CALL "Comp_errores"  ; Llamamos al módulo E/S para saber
                           ; si ha habido errores.
     CALL "FINOP_SI"      ; Informa al OS de que la rutina de
                           ; interrupción ha terminado.
     RET

```

Suponiendo:

$$100 \text{ MIPS} \rightarrow t_{\text{inst}} = \frac{1}{10^8} = 10 \mu\text{s} \quad V_{\text{transf}} = 10^6 \text{ b/s} \quad t_{\text{acc}} = 1 \text{ ms}$$

$$t_{\text{opES}} = t_{\text{ini}} + t_{\text{acc}} + \frac{1000}{10^6} + t_{\text{SRI}} + t_{\text{RSI}} + t_{\text{FIN}} = 1 \text{ ms} + 1 \text{ ms} = 2 \text{ ms}$$

$$t_r = 1000 (t_{\text{SRI}} + t_{\text{RSI}}) = 1000 (1 \text{ ns} + 15 \text{ ns}) = 160 \mu\text{s}$$

Es decir, cuatro veces más que en la programada. Ahora sabemos por qué se utiliza la técnica "por interrupciones":

$$t_{\text{CPULIBRE}} = 2000 - 160 \mu\text{s} = 1840 \mu\text{s}$$

$$\%CPU_{\text{LIBRE}} = \frac{1840}{2000} \times 100 = 92\%$$

Este porcentaje ya no se gasta en hacer bucles, sino que está ocupado en otros procesos, esa es la gran diferencia.

$$\%CPU_T = \frac{160}{2000} \times 100 = 8\% \text{ de gasto de la CPU}$$

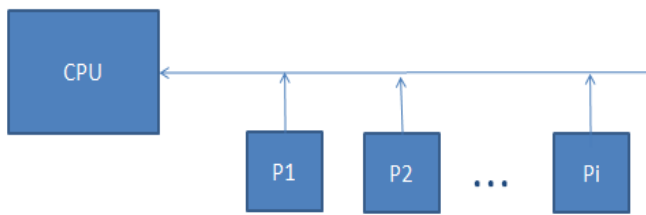
Problemas

- **Conexionado.** ¿Cómo conectar todos los módulos (ya que siempre hay más de uno) y sus respectivas líneas de petición de interrupción?
- **Identificación.** ¿Cómo se sabe a quién pertenece una interrupción?
- **Localización.** ¿Qué subrutina de servicio debe utilizarse en cada interrupción?
- **Prioridades.** ¿Cómo sabemos a quién dar prioridad cuando existen varios periféricos interrumpiendo?
- **Anidamiento de rutinas.**

Soluciones

Conexionado

La solución más utilizada y eficiente es conectar cada línea de interrupción en una sola hacia el CPU.



En cada conector existe una puerta llamada de colector abierta (en CMOS) o de drenador abierto (en NMOS) que permite que este cable de interrupción funcione como un OR cableado. Siempre que uno (o más) periféricos estén

activos el cable está activo y solo pasa a estar inactivo cuando TODOS los periféricos lo estén.

Identificación y localización

~mediante muestreo

Solo hay una rutina de tratamiento de interrupción (RTI) que pregunta uno a uno a todos los periféricos en el orden de prioridad dado. Cuando encuentre una petición de interrupción salta a la subrutina del módulo que la pedía, por tanto resuelve ya el problema de la prioridad además del de la localización e identificación. Sin embargo no permite anidamiento de rutinas y no es muy eficiente.

Vectorización

Esta forma de identificación conceptualmente dota a la CPU de una forma de preguntar quién está enviando la interrupción. Para ello hace uso de una nueva señal, en el ciclo de bus, de reconocimiento de interrupción (SRI), la señal INTA. Esta señal pide al dispositivo que se identifique. Para poder implementarlo es necesario que el módulo posea un vector de identificación.

Este vector debe ser sencillo de implementar.

Prioridades

A parte de las ya comentadas en el apartado anterior:

Gestor centralizado

Cada módulo tiene una señal de interrupción y una de reconocimiento. Así es sencillo conocer la prioridad y la identificación de cada módulo. Esto también permite tener flexibilidad en la asignación de prioridades. Sin embargo al estar el sistema cableado no es escalable, no pueden añadirse más módulos que los que ya existen.

También permite anidamiento, pero no es fácil de implementar.

Gestor encadenado

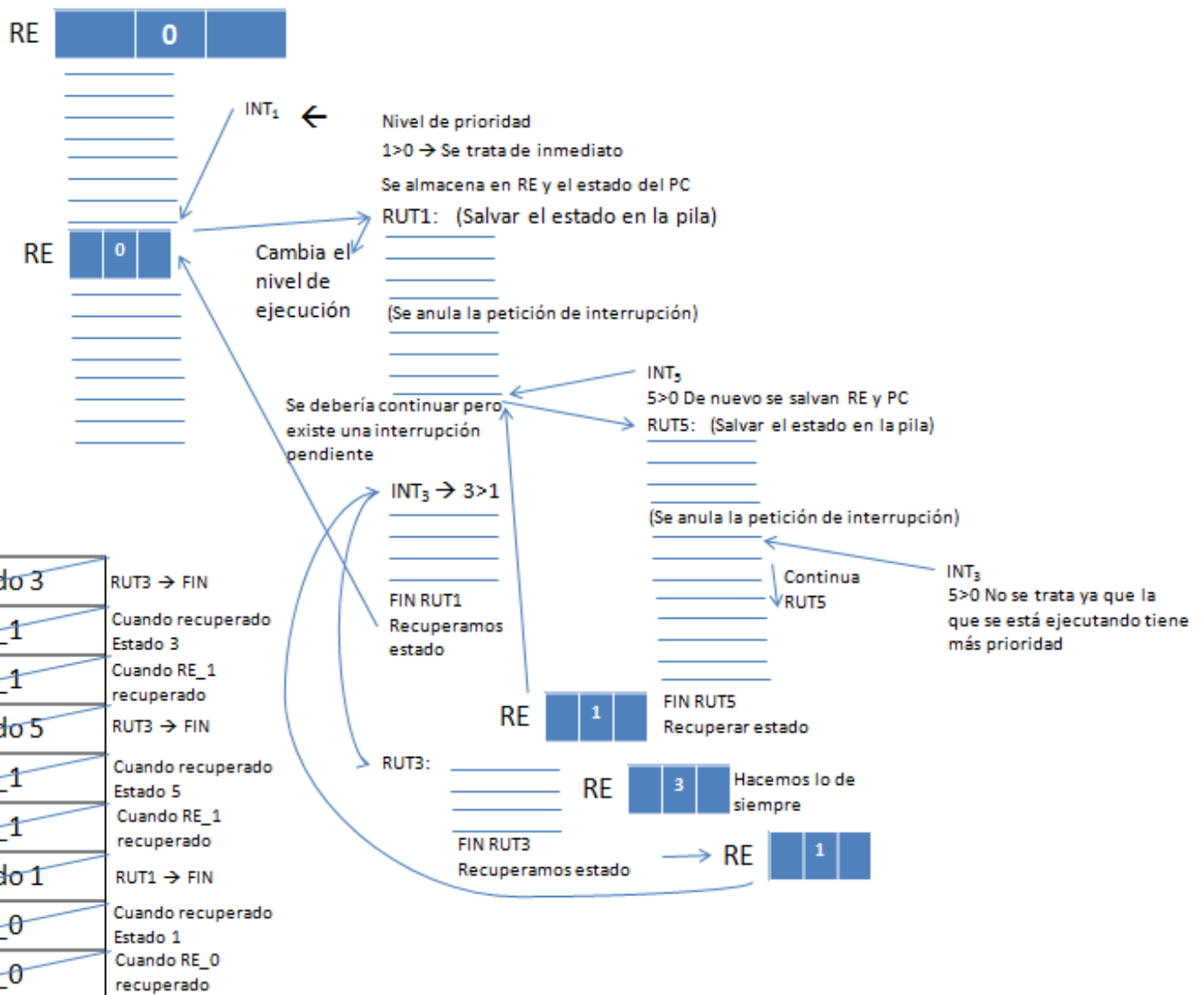
En este caso, la señal de reconocimiento es única y está conectada a todos los módulos de forma anidada. La señal pasa por todos los módulos en orden de prioridad y para en el primero que esté activo. La asignación de prioridades se establece por construcción y por tanto es fija, sin embargo, pueden añadirse más módulos.

Híbrido

Combina ambas soluciones.

Anidamiento

Ejemplo de ejecución



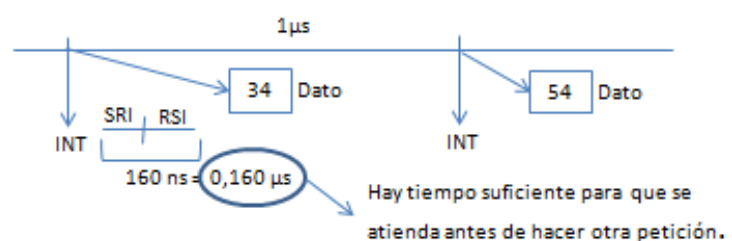
Cuando llega una petición de interrupción de nivel igual o inferior al proceso que se está ejecutando se evalúa únicamente cuando esta última ha terminado.

Si la petición es de nivel mayor se ejecuta de inmediato.

Ejercicio

Computador


- 10^6 b/seg
- 100 MIPS → $t_{ins} = 10$ ns
- $t_{SRI} = 1$ Instrucción
- $t_{RSI} = 15$ Instrucciones



Estado 3	RUT3 → FIN
Re_1	Cuando recuperado Estado 3
PC_1	Cuando RE_1 recuperado
Estado 5	RUT3 → FIN

$$\text{Frecuencia de interrupción} = \frac{10^6 b/s}{1 \text{ Ins}/B} = 10^6 \text{ Ins}/s$$

$$t_{\text{entre_ins}} = \frac{1}{10^6 \text{ Ins}/s} = 1 \mu s$$

$t_{\text{respuesta}} =$  $t_{\text{res_max}}$ → Es lo único que podemos saber.
Es el tiempo máximo que puede esperar una interrupción para no perder datos.

$$t_{\text{res_max}} = t_{\text{entre_ins}} - (t_{\text{SRI}} + t_{\text{RSI}})$$

↓
Pero puede ser que el CPU esté haciendo otra cosa o atendiendo otra petición y se posponga.

Puede incluso tratarse después de que llegue otra interrupción y se evalúe un dato distinto, perdiendo datos.

Existen, pues, límites al tratamiento por interrupciones que dependen, por un lado, de lo que tarda el periférico en emitir una interrupción. Por otro lado, de la velocidad a la que se procesen las instrucciones y el mínimo de instrucciones de la rutina RSI.

Consumo

$\text{Consumo}_{\text{CPU}} = \text{Frec}_{\text{int}} \cdot (t_{\text{SRI}} + t_{\text{RSI}}) = 10^6 \cdot 16 = 16 \text{ MIPS}$ es el consumo de IPS que gasta el periférico.

Si el consumo fuese mayor que las IPS que es capaz de ejecutar la CPU (en este caso, $\text{consumo}_{\text{CPU}} > 100 \text{ MIPS}$, no se da), el procesador no sería capaz de hacer funcionar el periférico.

Sumando el consumo de todos los periféricos tendremos el consumo total. Para que un sistema por interrupciones pueda funcionar, el consumo total debe ser menor que la velocidad del procesador.

$$\text{Cons}_{p1} = C_1 = Fl_1 \cdot (t_{\text{SRI}} + t_{\text{RSI}})$$

$$\text{Cons}_{p2} = C_2 = Fl_2 \cdot (t_{\text{SRI}} + t_{\text{RSI}})$$

·
·
·

$$+ \text{Cons}_{pn} = C_n$$

$$\Sigma \leq \text{MIPS}$$

Criterio de priorización

Existen múltiples criterios para asignar prioridades a las interrupciones, entre ellas las más importantes son:

- **Frecuencia de interrupciones.** Cuantas más interrupciones haga en un tiempo determinado más prioridad.

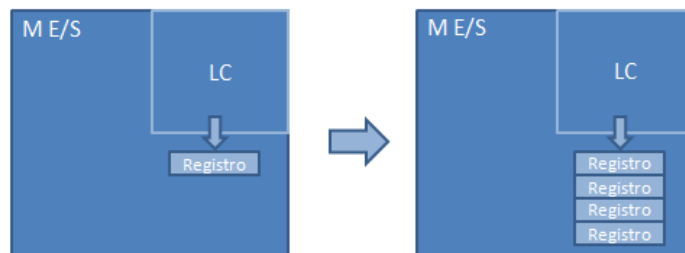
- **Urgencia de las instrucciones.** Periféricos que solo interrumpen en momentos críticos tienen alta prioridad. Se denominan interrupciones no enmascarables, para ellos existen peticiones de interrupción especiales.

Resolviendo problemas con las interrupciones

$$\begin{array}{lcl}
 t_{\text{respuesta}} \rightarrow & & V_{\text{transf}} \rightarrow \text{Depende del periférico. No podemos modificarla.} \\
 t_{\text{procesamiento}} \rightarrow & \left| \begin{array}{l} \text{Frec}_{\text{INT}} = \\ t_{\text{INT}} \text{ (Número de instrucciones necesarias para satisfacer las} \\ \text{interrupciones)} \rightarrow \text{Podemos modificarlo hasta cierto punto.} \end{array} \right. &
 \end{array}$$

Entonces, ¿Cómo podemos modificar un módulo de E/S y reducir su frecuencia?

Una de las cosas que podemos hacer es aumentar el número de bytes que se transfieren en cada operación. En vez de tener un registro donde se almacenan los datos tendremos varios registros.



Tenemos varias posibilidades:

- Número de registros que tengamos.
- Capacidad de cada registro.

Por ejemplo:

- 4x1 \rightarrow 4 registros de 1 byte \rightarrow 4 B
- 1x4 \rightarrow 1 registro de 4 bytes \rightarrow 4 B
- 4x4 \rightarrow 4 registros de 4 bytes \rightarrow 16 B

Cuanta más capacidad de memoria temporal tenga el módulo, más eficiente será el sistema.

Teniendo en cuenta este nuevo factor:

$$t_{opES} = t_{INI} + t_{acc} + \frac{N}{V_{transf}} + t_{int} + t_{FIN}$$

En este caso t_{int} será un poco más grande al ser la RSI más complicada. Sin embargo, t_{opES} apenas cambia.

$$t_{CPU} = t_{int} + \frac{N}{NR \times LR} \times (t_{SRI} + t_{RSI}) + t_{FIN}$$

Donde NR = Número de registros y LR = Longitud de registro.

Esta parte cambia considerablemente ya que disminuye el número de interrupciones, pues antes solicitaba una interrupción por cada byte y ahora cuando se llena la memoria temporal.

Existen módulos que envían dicha interrupción cuando reciben el primer byte. En este caso, el número de interrupciones es $\frac{N}{NR \times LR}$.

Tiempo de salida

$$t_{opW} = t_{INI} + t_{acc} + \frac{N}{V_{transf}} + t_{FIN}$$

En el caso de una operación W se termina cuando llega el último dato.

Acceso Directo a Memoria (DMA)

Básicamente la técnica DMA deja en manos del módulo de E/S, tanto la sincronización (como en la técnica por interrupciones) como la transferencia. Solo se avisa a la CPU al finalizar.

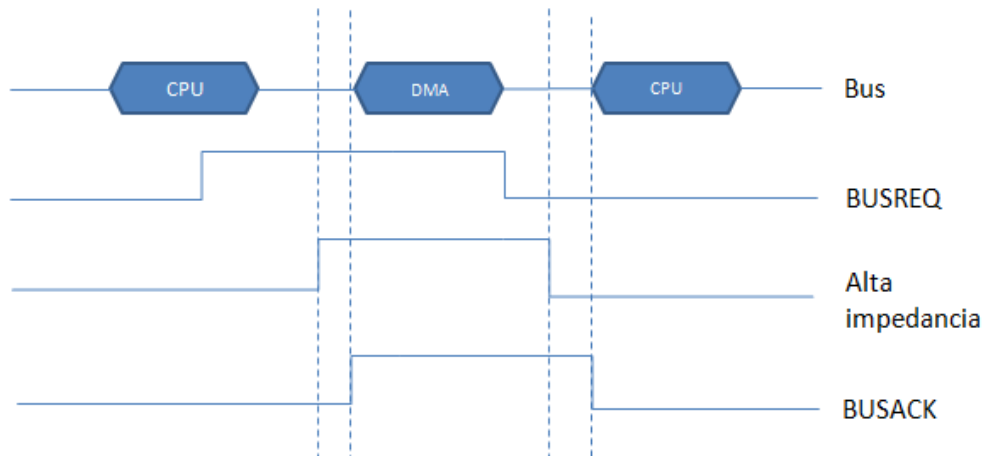
Para ello debemos realizar modificaciones en el módulo de E/S, hacerlo más complejo:

- Añadir dos registros más, el registro de direcciones y el registro de contador.
- Generar las señales de dirección y control (RD, WR, MEM) entre otras.
- Necesitamos un incrementador, un decrementador y un comprobador.
- Introduciremos dos señales nuevas, para solicitar el bus cuando el módulo necesite recibir o enviar información al mismo.

Petición de bus

Robo de ciclo aislado

Byte a byte. 1 registro de 1 byte.



$$t_{opES} = t_{INI} + t_{acc} + \frac{N}{V_{transf}} + t_{DMA} + t_{FIN}$$

$$t_{CPU} = t_{INI} + N \underbrace{(t_{PCD} + t_{CM})}_{t_{DMA}} + \underbrace{(t_{SRI} + t_{RSI})}_{FIN} \underbrace{(t_{SRI} + t_{RSI})}_{FIN}$$

Donde N es el número de operaciones de DMA: $N_{DMA} = \frac{N}{NR \times LR}$ y t_{PCD} es el tiempo del protocolo de concesión de datos.

Si el registro tiene una longitud mayor a un byte o tiene varios bytes, el número de operaciones de DMA será menor.

Robo de ciclo en ráfagas

Utiliza tantos ciclos de DMA como sean necesarios, reduciendo el número de interrupciones en el bus.



$$t_{CPU} = t_{INI} + \frac{N}{NR \times LR} \underbrace{(t_{PCD} + NR \times t_{CM})}_{t_{DMA}} + \underbrace{(t_{SRI} + t_{RSI})}_{FIN}$$

$$N^{\circ} DMA = \frac{N}{NR \times LR}$$

Si la operación es de salida, la primera operación de DMA se hace de forma paralela al t_{acc} .

Problemas

Ejercicio primero

Un procesador de 32 bits; 40 MIPS ; $t_{CM} = 20 \text{ ns}$

a)

SRI: FETCH: Si $i > \max \wedge I > RE.BMI$ (Donde, i = interrupción; \max = máxima de todas las INT; RE.BMI = Biestable de máscara de INT).

```

*PC → PILA
*RE → PILA
RE.BMI ← i
RE.S ← 1 ; Pasamos a modo supervisor
CRI: *INTA i ; BD → vector
*PC ← M(RBTv + vector x 4) ; RBTv (Registro Base de Tabla
; de Vectores.
(FETCH)
SINO: (FECH)

```

Todas las operaciones con asterisco son las que gastan tiempo, por tanto:

$$t_{SRI} = 4 \times 20 \text{ ns} = 80 \text{ ns}$$

b) Tenemos los datos:

- bloques = 64 → 1536 b $N = 1536 \text{ b} \rightarrow 1536 \cdot 8 \text{ bits}$

- RD = 32 bits $FR_{INT} \rightarrow \frac{32 \text{ bits}}{10^7 \frac{b}{s}} = 312500 \text{ Hz (también int por segundo)}$
- $V_{TRANSF} = 10^7 \text{ bits/s}$ $t_{TRANSF} = \frac{N}{V_{TRANSF}}$
- $t_{INI} = 50 \text{ l}$
- $t_{RSI} = 25 \text{ l}$

Se pide el número de instrucciones.

$$C_{CPU} = 312500 \times \left(\underbrace{80 \text{ ns}}_{T_{SRI}} + 25 \text{ Ins} \right) = 312500 (3,2 \text{ Ins} + 25 \text{ Ins}) = 8812500 \frac{I}{s}$$

$$= 8,8125 \text{ MIPS}$$

$$80 \times 10^{-9} \times 40 \times 10^6 = 3,2 \text{ Ins}$$

Entonces:

$$C_{CPULibre} = 40 \text{ MIPS} - 8,8125 \text{ MIPS} = 31,1875 \text{ MIPS}$$

Si se anula el t_{RSI} a 70 Ins y un procesador de 64 bytes con 10^8 b/s y velocidad de transferencia de bloques de 1024 b entonces:

$$FR_{INT} = \frac{10^8 \text{ bits/s}}{64 \times 8} = 195312,5 \text{ Hz}$$

$$C_{CPU} = 195312,5 \times (3,2 \text{ Ins} + 70 \text{ Ins}) = 14296875 \text{ Ins/s}$$

$$C_{libre} = 40 \times 10^6 - 14296875 = 25703125 \text{ Ins/s}$$

$$NI = \frac{1024 \times 8 \text{ bits}}{\underbrace{10^8 \text{ b/s}}_{t_{TRANSF}}} \times 25703125 = 2105 \text{ Ins}$$

Robo de ciclo aislado

Con 1536 B

$$\%t_{CPU} = \frac{t_{CPU}}{t_{opES}} \times 100$$

$$t_{opES} = t_{INI} + t_{acc} + \frac{1536 \times 8 \text{ b}}{\frac{10^9 \text{ b}}{s}} + t_{DMA} + (t_{SRI} + t_{RSI})$$

$$= 50 \text{ Ins} + 0 + \frac{1236 \times 8}{10^9} \times 40 \times 10^6 + (10 \text{ ns} + 20 \text{ ns}) \times 10^{-9} \times 40 \times 10^6$$

$$+ (3,2 + 70) \times \frac{1}{40 \times 10^6} = 15398 \mu s$$

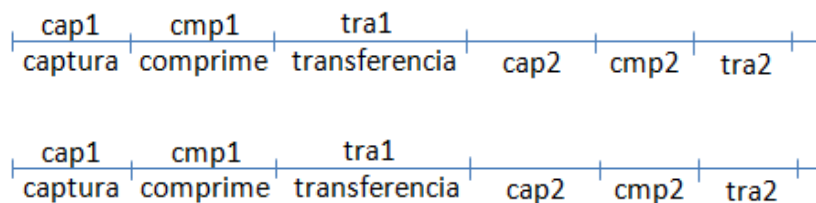
Ejercicio segundo

Computador

- 32 bits
- 400 MIPS
- Dedicado a la toma, compresión y transmisión de imágenes.
- Compresión de imágenes.
 - 512 KB → Se comprimen → 12 KB
 - El proceso de compresión dura $15 \cdot 10^6$ Ins
- Transmisión por internet
 - $V_{\text{TRANSF}} = 100 \cdot 10^6 \text{ bits/s}$
 - RD → 32 bits
 - Tamaño de bloque = 1,5 KB
 - Rutina INI → 100 Ins
- Captura de imágenes
 - $t_{\text{acc}} = 0$
 - $V_{\text{TRANSF}} = 40 \cdot 10^6 \text{ bits/s}$
 - RD → 32 bits
 - INI = 50 Ins

Nuestro objetivo es tratar de capturar 25 imágenes por segundo, ¿con qué técnica de E/S podríamos hacerlo?

E/S programada



Calculamos el tiempo (en instrucciones) que se tarda en capturar una imagen:

$$t_{\text{cap}} = t_{\text{INI}} + t_{\text{acc}} + \frac{t_{\text{amImg}}}{V_{\text{TRANSF}}} = 50 \text{ Ins} + 0 + \frac{512 \text{ KB}}{40 \times 10^6} = 50 + \underbrace{\frac{512}{40 \times 10^6} \times 400 \times 10^6}_{\text{Ins}} = 5242930 \text{ Ins}$$

Calculamos el tiempo (en instrucciones) que se tarda en transmitir una imagen:

$$t_{\text{tra}} = \frac{12 \text{ KB}}{1,5 \text{ KB}} \times \underbrace{\left(t_{\text{INI}} + t_{\text{acc}} + \frac{1,5 \times 8}{100 \times 10^6} \times 400 \times 10^6 \right)}_{49152 \text{ Ins}} = 394016 \text{ Ins}$$

Si sabemos que el tiempo de compresión (en instrucciones) es:

$$t_{\text{cmp}} = 15 \times 10^6 \text{ Ins}$$

El tiempo total será la suma de estos tres valores:

$$t_{ima} = t_{cap} + t_{cmp} + t_{tra} = 20636946 \text{ } Ins / Img$$

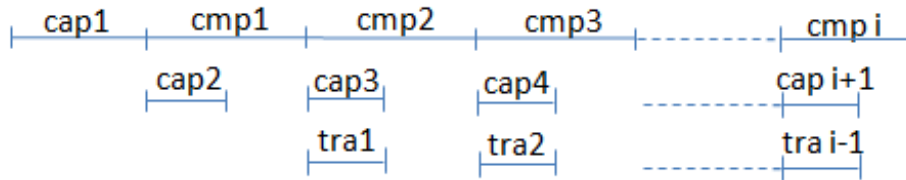
Entonces el procesador es capaz de procesar:

$$ImgSeg = \frac{400 \times 10^6}{20636946} = 19,38 \text{ } Img/s$$

Por tanto con una técnica de E/S programada no seríamos capaces de cumplir nuestro objetivo de procesar 25 imágenes por segundo.

E/S por interrupciones

En este caso mientras se comprime una imagen, se captura la siguiente y se envía la anterior. Suponemos una Subrutina de Tratamiento de Interrupción de 5 Instrucciones (SRI = 5 Ins), y una Rutina de Servicio de Interrupción de 20 Instrucciones en el caso de la capturadora y de 35 instrucciones en el caso de la transmisión (RSI_{cap} = 20 Ins y RSI_{tra} = 35 Ins).



Primero debemos determinar la prioridad de cada operación, que se calcula mediante su RSI y la frecuencia de interrupciones.

$$Frec_{intTra} = \frac{100 \times 10^6 \text{ bits/seg}}{32 \text{ bits}} = 3,125 \times 10^6 \text{ Hz } \left(\frac{\text{Interrupciones}}{\text{segundo}} \right)$$

$$Frec_{intCap} = \frac{40 \times 10^6 \text{ bits/seg}}{4 \text{ bytes}} = 10 \times 10^6 \text{ Hz}$$

También calculamos el número de instrucciones por segundo de cada proceso:

$$Consumo_{tra} = 3,125 \times 10^6 \times (5 \text{ Ins} + 35 \text{ Ins}) = 125 \times 10^6 \text{ Ins/s} = 125 \text{ MIPS}$$

$$Consumo_{cap} = 10 \times 10^6 \times (5 \text{ Ins} + 20 \text{ Ins}) = 250 \times 10^6 \text{ Ins/s} = 250 \text{ MIPS}$$

En total 375 MIPS serían necesarios para capturar y transmitir al mismo tiempo, menos que los 400 MIPS que es capaz de procesar.

Una vez conocemos esto, es necesario calcular el número de instrucciones de cada proceso para saber cuántas imágenes procesa por segundo:

$$t_{cap} = 50 \text{ Ins} + \frac{512}{4} \times (5 \text{ Ins} + 20 \text{ Ins}) = 3276850 \text{ Ins}$$

$$t_{tra} = \frac{12}{1,5} \times \left(100 \text{ Ins} + \frac{1,5}{4} \times (5 \text{ Ins} + 35 \text{ Ins}) \right) = 123680 \text{ Ins}$$

$$t_{cmp} = 15 \times 10^6 \text{ Ins}$$

En total:

$$t_{tot} = 18400030 \text{ Ins}$$

Por lo tanto:

$$\frac{400 \times 10^6}{18,400030 \times 10^6} = 21,73 \text{ Img/s}$$

Con una técnica de E/S por interrupciones seguimos sin cumplir nuestro objetivo de procesar 25 imágenes por segundo.

E/S por DMA

Suponiendo que $t_{DMA} = 10 \text{ ns}$ que son $10 \times 10^{-9} \times 400 \times 10^6 = 4 \text{ Ins}$. Entonces:

$$\begin{aligned} t_{cap} &= 50 \text{ Ins} + \frac{512}{4} \times 10 \text{ ns} + (5 \text{ Ins} + 20 \text{ Ins}) \\ &= 50 \text{ Ins} + \frac{512}{4} \times 4 \text{ Ins} + (5 \text{ Ins} + 20 \text{ Ins}) = 524363 \text{ Ins} \end{aligned}$$

$$t_{tra} = \frac{12}{1,5} \times \left(100 \text{ Ins} + \frac{1,5}{4} \times 4 + (5 \text{ Ins} + 35 \text{ Ins}) \right) = 13408 \text{ Ins}$$

$$t_{cmp} = 15 \times 10^6$$

En total:

$$t_{tot} = 1537771 \text{ Ins}$$

Si necesitamos procesar 25 imágenes por segundo, ¿podemos utilizar DMA?

$$25 \times 1537771 = 388444275 \text{ Ins/s}$$

Que es menor de 400, por tanto: sí, sería posible. Sin embargo, observamos que el consumo del procesador con estas operaciones es:

$$\frac{388444275}{400 \times 10^6} \times 100 = 97,11 \%$$

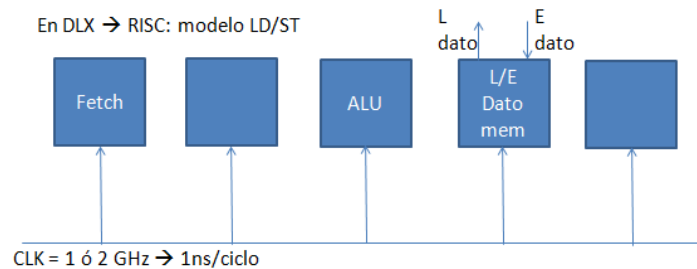
Que es una cantidad excesivamente alta, este procesador estaría al límite de su capacidad. Por tanto sería recomendable tener un procesador capaz de procesar más instrucciones.

Tema 2 – Sistema de memoria

Profesor: Luis M. Gómez Henríquez

Introducción

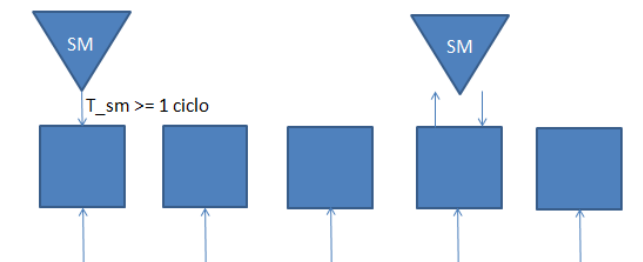
Pipeline de instrucciones



En algún momento de la vida de un programa debemos acceder a memoria. El problema es que el acceso a memoria es mucho más lento que la velocidad del procesador y el pipeline se llena y eso es inviable para un computador estable.

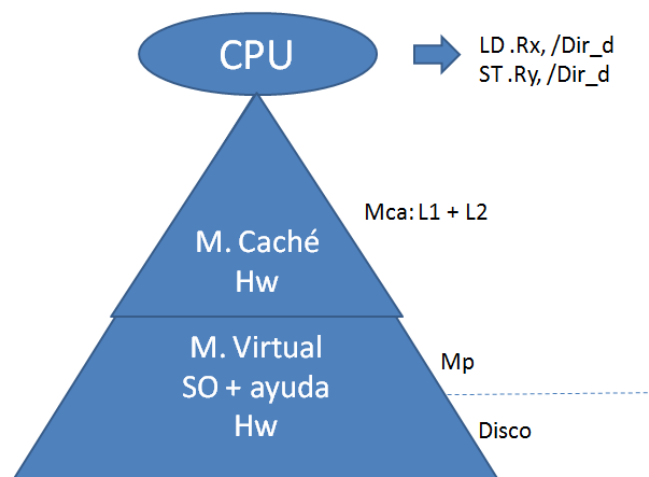
Para solucionar este problema se ideó el sistema de memoria, mucho más rápida en tiempo de acceso. Lo ideal es que tarde aproximadamente un ciclo.

Esto se conseguirá con una jerarquía de memoria, pequeñas y de rápido acceso que sean los verdaderos interlocutores entre la memoria principal y el procesador. A cada nivel se le denomina nivel de caché. El primer nivel es el más pequeño del orden de 32 o 64 KB, pero en la que se encuentra el 95% de los accesos.



Jerarquía de memorias (JM) y Memoria cache

Ubicación automática de dirección



mecanismo es transparente a la CPU.

Desde la CPU se envían direcciones que primero se buscan en la caché L1 (donde se encuentra en el 95% de los casos), si no se encuentra se busca en L2, si no se encuentra ahí se busca en memoria virtual (aunque hay ordenadores con caché L3) y, por último, en memoria principal. Si no se encuentra aquí ocurre una catástrofe ya que se debería buscar en memorias físicas 100000 veces más lentas que la memoria principal. Por eso es importante el sistema de memoria. Este

Principio de inclusión

Cada nivel contiene un subconjunto de las direcciones que se encuentran en niveles superiores. Se cumple el principio de inclusión. Si D_j es el conjunto de instrucciones de nivel j , se cumple que:

$$\dots \subset D_{j-1} \subset D_j \subset D_{j+1} \subset \dots$$

Aunque existen excepciones.

Funcionamiento

Cuando la CPU solicita acceso al nivel 1 comprueba si tiene la información solicitada, si la tiene se sirve a la CPU. Si no la tiene se comprueba el nivel 2, si tiene la información se transfiere a nivel 1 y el nivel 1, a su vez, se sirve a la CPU. Si no se tuviese en caché se efectuaría una operación en M_p y, si no posee la información, entra en juego el sistema operativo que pausa el proceso hasta que se obtiene la información.

Siempre se copia un conjunto de direcciones consecutivas: bloques de caché (de unos cuantos bytes) o páginas (del orden de KB).

Motivación

Ya hemos comentado que la caché L1 contiene el 95% de las instrucciones. Esto se debe, también, a la forma de trabajar de los programas que suelen hacer referencia a direcciones y datos consecutivos.

Existe una existencia de direcciones que se repiten a lo largo de un programa, que son próximos en el tiempo. También podemos observar una tendencia a una proximidad espacial, casi consecutivas.

Políticas

Política de ubicación

Ej: $B_q = 2^4 \text{ B/bloque}$
 $C = 8 \text{ KB}$

$c = 512 \text{ líneas}$

C = capacidad total de la M_{ca}

B_1 = tamaño de los bloques. Siempre potencia de 2.

$$M_p \rightarrow M = 4 \text{ GB} \rightarrow 32 \text{ bits}$$

Dependiendo de la política de ubicación. La CPU solo ve 32 bits, una palabra al tiempo:



Si tomamos 4 bloques, que corresponde al *truco*, quedan 28 bloques.

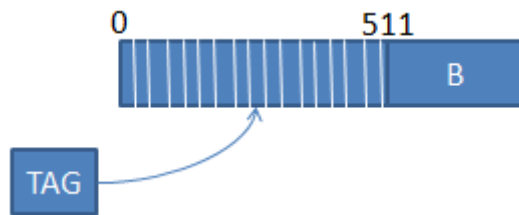
Completamente asociativa

Cualquier bloque puede ponerse en cualquier línea. No se utiliza por ser demasiado lenta.



Directa

Cada bloque va directamente a su línea correspondiente por orden de su módulo.

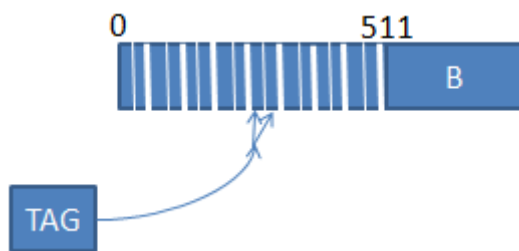


Asociativa por conjuntos (por sets)

Posee un grado de libertad, S (2 ó 4) que indica el número de líneas por set.

$$s = c/S \text{ (s = nº de sets)}$$

Cada bloque se coloca en la primera posición libre de su set correspondiente por determinada operación. Por ejemplo, $i \bmod s$.



Política de extracción

Decide cuándo y qué bloques se llevan desde Mp a Mca.

Bajo demanda

El controlador de la caché trae a Mca el bloque que produzca un fallo (por no estar). No se utiliza en la vida real, pero es muy típico en exámenes y ejercicios.

Con anticipación (prefetching)

Se fundamenta en el fenómeno de proximidad espacial. Trata de llevarse a Mca bloques que, previsiblemente, se van a necesitar en el futuro. Mejora la tasa de aciertos en media.

Siempre (Always prefetch)

Si referencia a bloque i → subir bloque i+1 → Siempre que no se haya subido ya.

Ante un fallo (On a miss)

Si referencia y fallo en bloque i → subir {bloque i, bloque i+1}

Etiquetada (tagged)

Si referencia y fallo en bloque i → subir {bloque i, bloque i+1* (marcado con *)},...

Si referencia a un bloque j* → subir {bloque j+1*} desmarcamos {bloque j}

Es decir, cuando hay un fallo en el bloque i se sube este y su siguiente, este último se marca con una etiqueta. Por tanto, todos los bloques subidos en prefetching están marcados. Cuando se referencia a un bloque marcado nuestra intuición anterior era acertada y se desmarca y se sube el siguiente marcado.

Política de reemplazo

Selecciona qué bloque se desaloja de Mca para albergar a uno que se sube de Mp. Solo es necesaria en cachés no directas.

Aleatoria

Es muy difícil de implementar.

FIFO

El más antiguo se reemplaza.

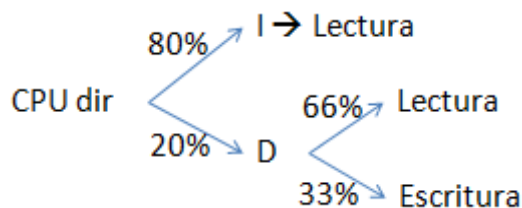
LRU

Least Recently Used. Se reemplaza el bloque al que no se ha accedido desde hace más tiempo.

El problema con las dos últimas políticas es que requieren que Mca almacene información adicional.

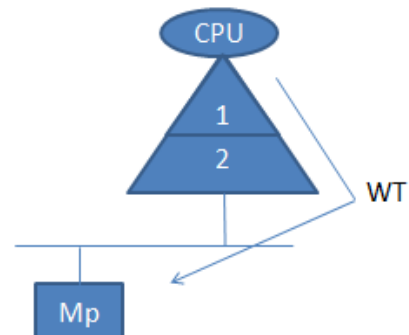
Política de escritura

Si hay acierto en la escritura en Mca, ¿Cuándo se actualiza la información en el siguiente nivel?



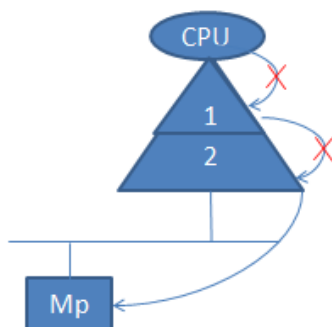
Escritura inmediata (*write-through* o WT)

Se escribe en Mca y en el siguiente nivel con coherencia entre niveles adyacentes.



Escritura aplazada (*Copy-Back* o CB)

Se escribe solo en la Mca y se actualiza en el siguiente nivel solo cuando es reemplazado.



En la vida real se mezclan ambas políticas, WT entre niveles interiores de la caché y CB entre la caché y la Mp.

Cuando hay fallo en escritura:

Sin copia en Mca (*with no allocation* o WNA)
El bloque NO se sube.

Con copia en Mca (*with allocation* o WA)
El bloque SÍ se sube.

Tamaño de la Mca y de sus bloques

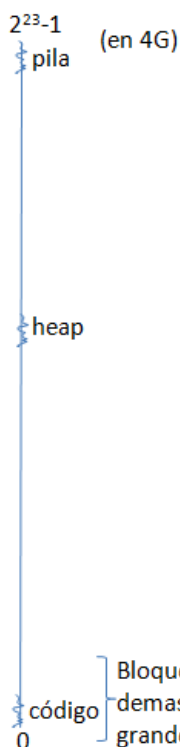
El tamaño de la Mca (C) y de sus bloques (Bq) influyen tanto en la tasa de aciertos (Hr) como en el tiempo medio de acceso:

$$C \uparrow \rightarrow Hr_{Mca} \uparrow \text{ (pero) } t_{ej} \uparrow$$

Por eso las cachés no son demasiado grandes, lo importante es el tiempo de acceso:

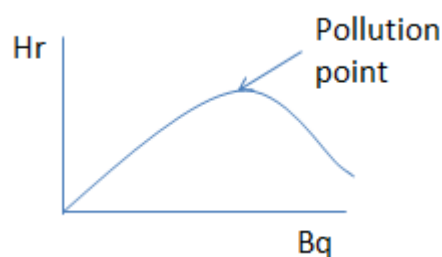
$$Bq \uparrow \rightarrow Hr_{Mca} \uparrow \text{ (hasta cierto momento donde } Hr_{Mca} \downarrow \text{)}$$

Para comprender mejor este hecho veamos el uso de las direcciones de un proceso; lo que se llama imagen de un proceso:



Como vemos, las zonas usadas son subconjuntos muy pequeños del espacio total de direcciones. Son zonas disjuntas y tienen un tamaño que varía dinámicamente. Entonces, llega un momento en el que el tamaño de los bloques es más grande que las direcciones efectivas.

A partir de este punto, denominado *pollution point*, existen más direcciones erróneas que correctas.



Problema 1

	ORG 0	Dir
	LD .R3, #0	0
	LD .R1, #10	4
	LD .R2, #1024	8
ETI:	ADD .R3, [.R2]	12
	ADD .R2, #4	16
	SUB .R1, #1	20
	BNZ \$ETI	24

- 4 B/inst
- Dir. a nivel de byte (B)
- Mca: → I+D (unificada), un solo nivel (L1)
 - Asoc. por sets o conjuntos
 - 256 sets
 - 2 bloques/set = S (grado de libertad)
 - 16 B/bloque = 2^4 B/bloque = Bq

Primera parte

Traza

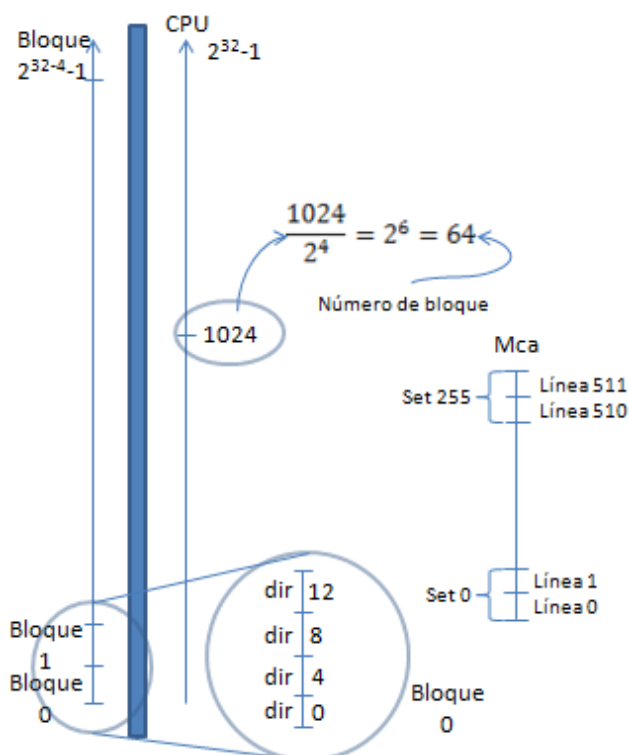
10 iteraciones del bucle

Dirs	(I)	0	4	8	12		16	20	24	12		16	20	24	12		...		16	20	24
(D)					1024					1028					1032		...	1060			

El único acceso a memoria está en la instrucción de la dirección 12. A partir de ahora para el resto del ejercicio solo nos interesa esta traza.

Segunda parte

Evolución de la Mca y la Hr_{Mca}



Ubicación: Bloque $i \rightarrow \text{set } i \bmod s$

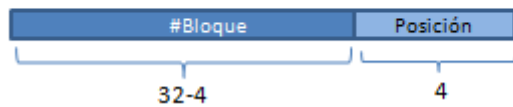
Tenemos 4 (palabras o direcciones)/bloque. Entonces, en cada bloque pondremos 4 direcciones de la traza.

$$\underbrace{0, 4, 8, 12}_{\text{Bloque 0}} \quad \underbrace{16, 20, 24, 28}_{\text{Bloque 1}}$$

Aunque 28 no pertenezca a la traza los bloques siempre deben estar completos.

Cada vez que queramos acceder a una instrucción se accederá al bloque de la Mca donde se encuentre dicha dirección.

bits 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 =1024
31 30 ... 10 9 8 7 6 5 4 3 2 1 0



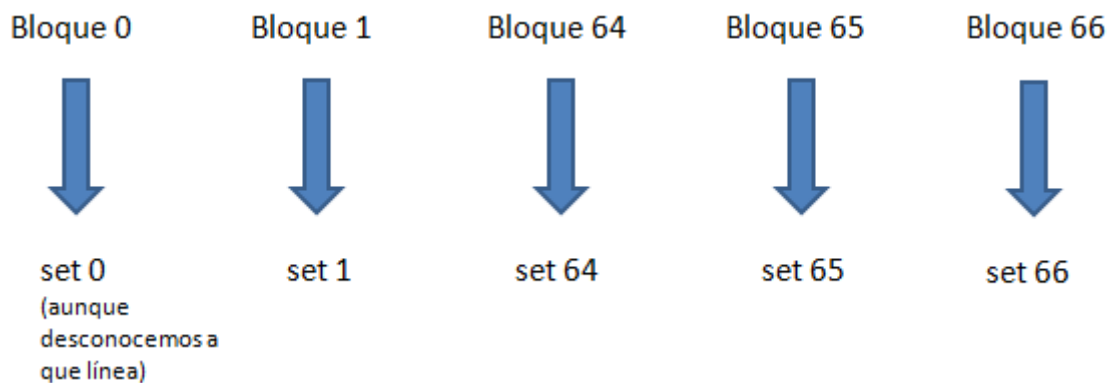
Puesto que el dato ya estaba en memoria podemos conocer el bloque en el que está mediante la fórmula:

$$\frac{1024}{2^4} = 2^6 = 64$$

1024, 1028, 1032, 1036
Bloque 64

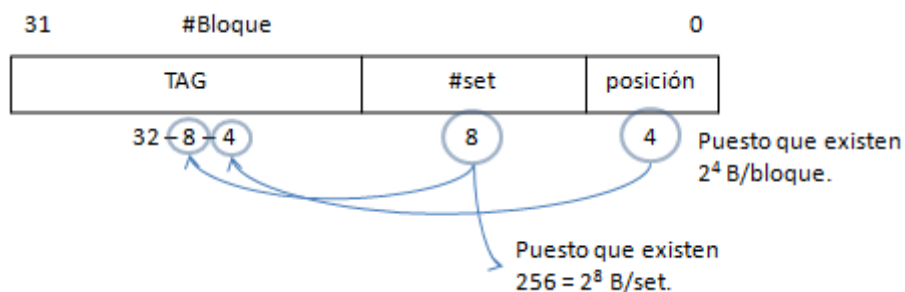
Los siguientes bloques, 65 y 66, serán los siguientes bloques de datos.

Ahora debemos estudiar como cada bloque se introduce en la cache. Debemos seguir la política de ubicación que:



Nota

Si siguiésemos una política completamente asociativa:



Si fuese una política directa:

31	#Bloque	0
TAG	#línea	posición
19	9	4

La primera vez que se intenta acceder al set 0 se produce un fallo de cache ya que no se encuentra el bloque 0. Por tanto, el primer acceso tardará un poco más que el tiempo de cache puesto que tiene que subir al set el bloque entero. Los siguientes accesos al set son correctos. Después se accede (o se trata de acceder) a la dirección 1024, en el bloque 64 que debería estar en el set 64, como no está todavía el acceso da error así que, como antes, se sube el bloque y, por tanto, aumenta el tiempo de acceso. Los siguientes no darán fallo. Así todos los bloques darían error en su primer acceso porque no tenemos anticipación. En este caso, en total, 5 errores.

Diagrama de fallos:

```

    #fallos
    /   \
  3 fallos D   2 fallos I
    \   /
      5 fallos
  
```

$$Hr_{Mca} = \frac{\#aciertos - \#fallos}{\#accesos} = \frac{53 - 5}{53} = \frac{48}{53} = 0,9056$$

#accesos $\rightarrow 3 + 4 \cdot 10 + 10 = 53$ accesos $Hr_{Mca} \approx 91\% (90,56\%)$

I
D

Tercera parte

Teniendo en cuenta que:

$$t_{SM} = t_{ef} = t_{acc}$$

$$t_{Mca} = 2 \text{ ns}$$

$$t_{fallo} = 42 \text{ ns}$$

Calculamos:

$$t_{ef} = \underbrace{Hr_{Mca} \times t_{Mca}}_{\text{acierto}} + \underbrace{(1 - Hr_{Mca}) \times (t_{Mca} + t_{fallo})}_{\text{fallo}} = t_{Mca} + (1 - Hr_{Mca}) \times t_{fallo}$$

$$= 2 \text{ ns} + (1 - 0,9056) \times 42 \text{ ns} = 5,96 \text{ ns/acceso}$$

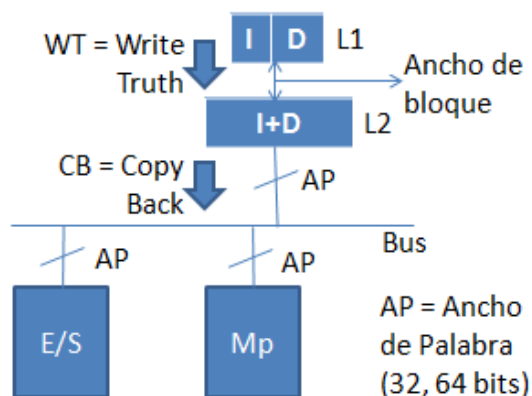
Efectuado de otra manera:

$$t_{SM} = \frac{53 \times t_{Mca} + 5 \times t_{fallo}}{53 \text{ accesos}} = 5,96 \text{ ns/acceso}$$

Memorias cache separadas

Los accesos a direcciones de datos (D) y a instrucciones (I) presentan un comportamiento diferente. Hasta ahora hemos trabajado con una única cache pero existe una manera de separar estos dos accesos en dos caches diferentes. Esto se denomina arquitectura Harvard. Es una forma de mejorar la eficiencia en el acceso.

Memoria cache multinivel



Entre la cache L1 y L2 la comunicación se establece a nivel de ancho de bloque. En los ciclos necesarios, se sube todo un bloque. El nivel 1 de cache está fragmentado, mientras que el nivel 2 siempre es unificado.

En los últimos años la cache ha ganado un nuevo nivel, L3, aunque únicamente en los procesadores multicore.

Normalmente se cumple el principio de inclusión, es decir, que el nivel 1 es un subconjunto del nivel 2.

Ambos niveles tienen diferentes consideraciones en su diseño, en el nivel 1 tratamos de reducir el tiempo de acierto, en el nivel 2 lo que queremos reducir es la tasa de fallos y reducir el $t_{\text{penalización}}$. En la siguiente tabla podemos observar las diferencias:

	Tamaño	T_{acierto}	Asoc.	Unificada
L1	16 - 64 kB	1 - 3 ciclos	1 - 2	No
L2	256 kB - 2 MB	7 - 15 aciertos	8 - 16	Sí
L3	2 MB - 8 MB	20 - 30 ciclos	16 - 32	Sí

El objetivo de la cache L2 es satisfacer, lo más rápido posible los fallos de la cache L1.

Medidas de rendimiento

Tasa de aciertos (Hr_{Mca})

$$\frac{N^{\circ} \text{ accesos con aciertos en } Mca}{N^{\circ} \text{ total de accesos}} \approx 100\% \rightarrow \text{Objetivo ideal}$$

Tiempo medio de acceso (T_{ef})

$$\frac{T_{\text{acierto}}}{t_{\text{cache}}} + (1 - Hr) \times T_{\text{penalización}} \approx T_{Mca} \rightarrow \text{Objetivo ideal}$$

Es el tiempo transcurrido desde que la CPU realiza una petición al SM hasta que la CPU pueda continuar (por ello también se denomina T_{SM} o T_{acc}).

Tiempo medio de ocupación (T_{ocup})

$$T_{ef} + T_{act. Mca} \approx T \text{ entre peticiones de la CPU} \rightarrow \text{Objetivo ideal}$$

Es el tiempo transcurrido desde que el SM sirve una petición hasta que puede servir la siguiente. Normalmente T_{ef} y T_{ocup} son iguales, tan solo en algunos casos y en ciertas arquitecturas de la cache pueden diferenciarse.

En caches multinivel

Para Mca L2 (y, en su caso, L3):

Si hay fallo Mca L1 y acierto en Mca L2 $\rightarrow t_{acc} \ll t_{MP}$

$$Hr_{local} = \frac{n^{\circ} \text{ aciertos en McaL2}}{n^{\circ} \text{ accesos a McaL2}}$$

$$Hr_{global} = \frac{n^{\circ} \text{ aciertos en McaL2}}{n^{\circ} \text{ accesos de la CPU}}$$

En L1 el Hr local coincide con el Hr global, por eso se hace la diferencia de caches.

Ejemplo

L1: $Hr_{McaL1} = 0,94$ L2: $Hr_{Local} = 0,60$

Por tanto, si añadimos a la cache L1 la cache L2 tenemos:

$$Hr_{Mca} = Hr_{L1} + (1 - Hr_{L1}) \times 0,6 = 0,94 + (1 - 0,94) \times 0,6 = 0,97$$

Lo cual es solo un 3% más. Sin embargo, la utilidad más interesante de la cache L2 es que es mucho más veloz que si la cache L1 comunicara directamente con la Mp.

Sin embargo, en L1 existen dos Hr_{local} , el Hr de su fragmento de datos (más pequeño) y otro de instrucción:

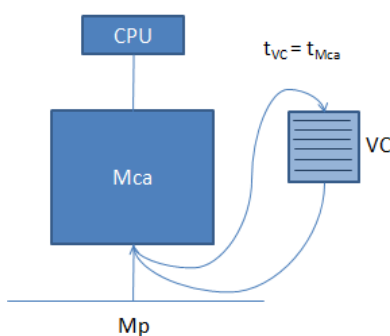
%I = 0,80 %D = 0,20

$Hr_{L1D} = 0,92$

$$Hr_{L1} = 0,80 \cdot 0,94 + 0,20 \cdot 0,94 = 0,936 \approx 0,94$$

Algunas mejoras

Victim cache



Almacena temporalmente bloques desalojados de Mca \rightarrow Si se demanda y se encuentra en la VC no hace falta traerlo de Mp. Muy útil en fallos por conflicto.

Se utiliza mucho, sobre todo conectado directamente a cache L1. Se utiliza cuando la política de ubicación es poco asociativa.

Lectura fuera de orden

Es una de las situaciones en las que se distingue t_{ef} de t_{ocup} . Fuera de orden (out of order, OOF) es muy habitual en los problemas.

Buffer de escritura (WB)

Suele estar asociado a Write Truth (WT), trata de atenuar el tiempo de escritura con acierto, que es igual al tiempo de acceso a memoria. Cuando tenemos WB y la Mca trata de escribir con acierto si tenemos buffer de escritura, el tiempo pasa a ser igual al tiempo de WB: $t_{SM} = t_{WB}$

Caches no bloqueantes

Cuando se ha servido una dirección que ha producido fallo y mientras algo pasa, no bloquea la CPU.

Tipos de fallo en memorias cache

Son básicamente tres:

- **Fallos de primera referencia.** Si no existe *prefetching* este fallo es inevitable ya que es el fallo que se produce cuando se hace referencia por primera vez a una dirección. También se denominan fallos en frío. Pueden ser calculados, en su mayor parte, antes de ejecutar.
- **Fallos por conflicto.** Este error se da cuando la política de ubicación nos obliga a insertar un bloque en un lugar ya ocupado de la Mca.
- **Fallos de capacidad.** Este error se da cuando la memoria está llena.

Inicialización y borrado

Al iniciar, las etiquetas de las entradas del directorio siempre tendrán un valor. En cada uno se añade un bit de validez que indica si esa entrada es válida o no:

$$\frac{V(\text{bit de validez})}{\text{línea}}$$

La instrucción *Flush_cache* es una instrucción de la arquitectura que invalida la cache entera. Invalidar es la forma de borrar una entrada de la cache. Cuando un nivel (o Mp) sube un bloque siempre lo coloca en una entrada marcada como inválida y cambia su bit de validez.

Ejercicios de examen

Ejercicio 1

Apartado a

OOF (*Out Of Order fetch*) → Reduce $t_{\text{penalización Mca}}$ → OOF → fallo x (Escritura y lectura) → baja
T espera en la CPU = T_{SM}

$T_{\text{ocupación}} \text{ VS } t_{SM} \leftarrow \text{OOF afecta solo a } T_{SM}$

Apartado b

El tamaño de los bloques influye en:

- $Hr_{Mca} \rightarrow$ Mejora hasta cierto punto (*pollution point*) donde baja.
- $t_{\text{penalización}} = t_{\text{fallo}} \rightarrow \underbrace{t_{\text{subir Bloque}}}_{OOF} \underbrace{(+t_{\text{bajar Bloque}})}_{CB} \rightarrow$ aumenta cuanto mayor sea el bloque.

Ejercicio 2

Tenemos un computador que:

- $w = 32$ bits (ancho de palabra)
- $Mp = 4GB$
- Mca unificada (I+D)
- Un nivel L1 (por defecto)

- L1: $C = 64 \text{ KB/cache} \rightarrow 2^{16} \text{ B/cache}$
- Asociativa por conjuntos de 4 bloques: $S = 4 \text{ bloques/set} = 2^2 \text{ B/set}$
- $Bq = 64 \text{ B} \rightarrow 2^6 \text{ B/bloque}$

Apartado a

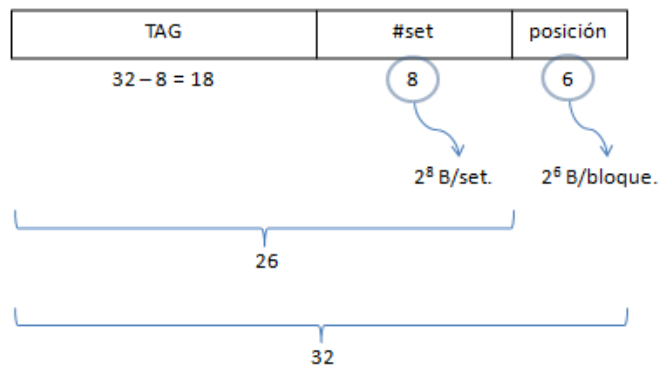
Deducciones

- Número de líneas, $c = \frac{2^{16} \text{ B/cache}}{2^6 \text{ B/cache}} = 2^{10} \text{ bloques o líneas en la cache}$
- Número de sets, $s = \frac{2^{10} \text{ líneas/cache}}{2^2 \text{ líneas/set}} = 2^8 \text{ sets/cache}$

Apartado b

En esta política de ubicación el bloque tiene la siguiente estructura:

Bloque $i \rightarrow 4 \text{ líneas en el set } i \bmod s$



Apartado c

$t_{MP} = 50 \text{ ns}$

$t_{\uparrow \downarrow \text{bloque}} = 132 \text{ ns}$

- entrelazado.
- Operación op. de ráfagas (Buses).

$$Bq = 64 \text{ B/bloque} \rightarrow \frac{\frac{64 \text{ B}}{4 \text{ B}}}{\text{bloque}} = 16 \frac{\text{pal}}{\text{bloque}}$$

Apartado d

Queremos conocer t_{SM} , t_{ef} , t_{acc}

- $t_{Mca} = 2 \text{ ns}$ (siempre pasa por la cache, sea cierta o fallo).
- OOF(política de escritura).
- CBWA
 - **Acierto.** Sin problema.
 - **Fallo.**
 - El bloque se reemplaza si modificado $\rightarrow \downarrow \text{Bloque} + \uparrow \text{Bloque fallo}$
 - El bloque no se reemplaza si no modificado $\rightarrow \uparrow \text{Bloque fallo}$
- Cuando se reemplaza un bloque el 25% está modificado.

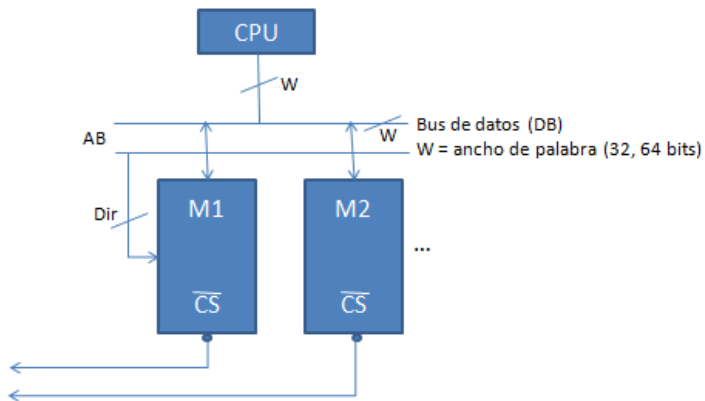
En CBWA no se diferencia lectura de escritura porque su tiempo es idéntico.

$$t_{SM} = t_{Mca} + (1 - Hr_{Mca}) + t_{fallo}$$

$$t_{SMmin} = t_{Mca}$$

$$t_{SMmax} = t_{Mca} + t_{\downarrow bloque} + t_{leer Pal} = 2 ns + (si mod) 123 ns + 50 ns$$

Memoria principal



Con la parte baja de la dirección se direcciona la pastilla Mi. con los bits de mayor peso se selecciona que chip se activa con un decodificador de la memoria. Sin embargo, esta estructura de una memoria no se ve, ni se trabaja con ella. Se utiliza su visión como un solo bloque.

t_{MP} es el tiempo que tarda en leer

o escribir una palabra de W bits en Mp.

Ancho de banda es el nº de bytes que se pueden transmitir a/desde la Mp.

Lo ideal es tener un mayor ancho de palabra. Para ello podemos utilizar la técnica de entrelazado:

- Sin entrelazado = $t_{s/b \text{ bloque}} \approx Bq \cdot t_{MP} + t_{Mca}$
- Con entrelazado = $t_{s/b \text{ bloque}} \approx t_{MP} + t_{Mca}$

Entrelazado

Entrelazado inferior

0	4	8	12
16	20	24	28
M0	M1	M2	M3

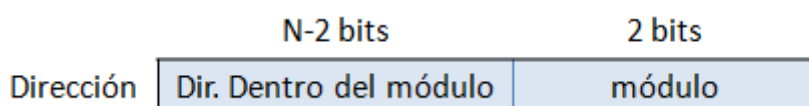
$$K = 4 = 2^2$$

Se trata de organizar la Mp como K módulos ("de orden k") a los que se puede acceder simultáneamente. K es una potencia de 2.

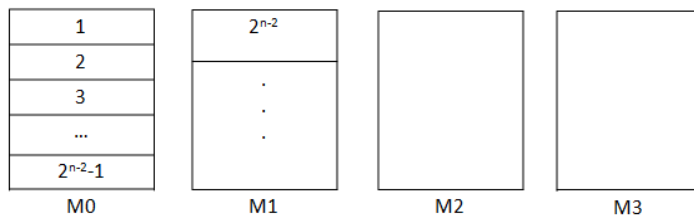
Con este sistema conseguimos que en el caso mejor se puedan

leer todas las primeras direcciones en un solo acceso.

Se direcciona como decíamos antes, mediante una parte de la dirección:

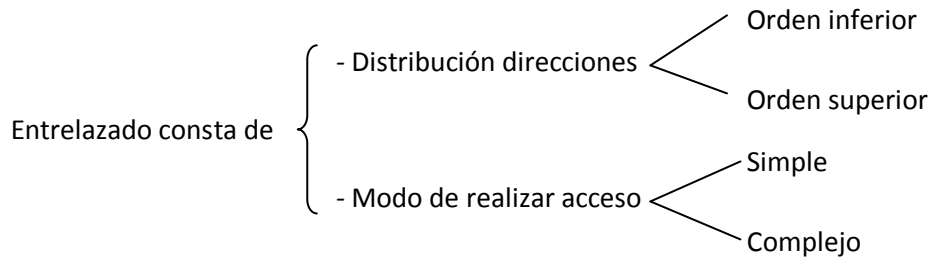


Entrelazado superior



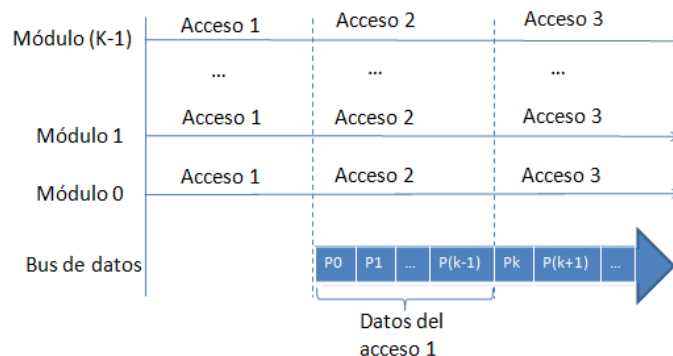
La diferencia es que los módulos tienen sus direcciones consecutivas. Se direcciona al módulo al revés, los primeros bits para el módulo y los finales para la dirección interna. Este modo es

muy engorroso y se usa poco.



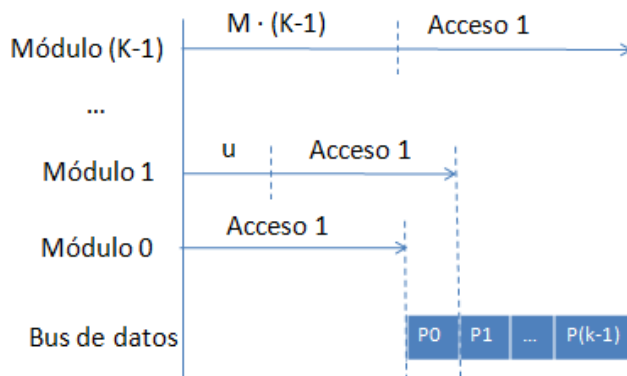
Entrelazado simple

Junto con orden inferior. La idea es que tenemos al menos un registro de datos por módulo donde se cargan las palabras de la dirección pedida, en un tiempo de acceso se cargan las palabras. El tratamiento es el mismo hacia arriba, con la diferencia de que solo puede escribirse desde el bus en un registro por acceso y luego escribirlos a la vez en cada módulo



Entrelazado complejo

Este método nunca se utiliza.



Ejercicios de examen

Ejercicio 1

¿En qué consiste la JM? ¿Cuáles son sus componentes? ¿Cómo funciona? Y discuta las políticas de ubicación y reemplazo.

La jerarquía de memoria es el orden de acceso a la memoria, en función de la velocidad. El nivel más bajo tiene una velocidad de acceso similar a la de la CPU, pero muy poca capacidad. Según se sube en la jerarquía aumenta la velocidad y la capacidad.

Está compuesta por una memoria principal y una memoria física en la parte superior de la jerarquía, más abajo la memoria cache (dividida en dos o tres niveles).

[INCOMPLETO]

Ejercicio 2

Apartado A

Mp entrelazada Es una técnica software para mejorar el Hr_{Mca} : **FALSO**, ni es una técnica software ni sirve para mejorar el Hr_{Mca} .

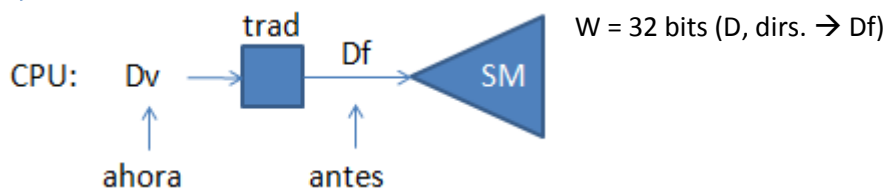
Corrección: Es una técnica hardware para disminuir t_{SM} , es decir, el tiempo que se tarda en subir y bajar un bloque.

Apartado B

En un sistema de memoria de lectura con Mca el tiempo invertido es independiente de la política de escritura con Mca: **FALSO**

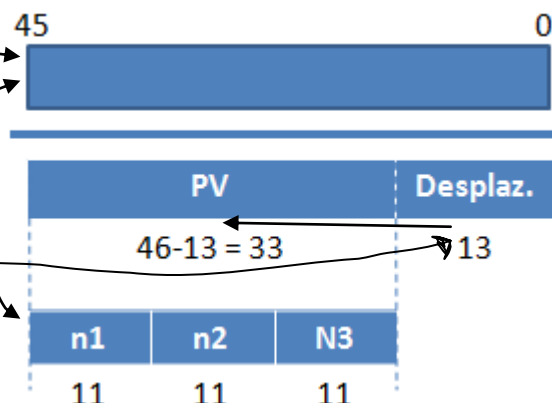
Corrección: Sí hay fallo si se modifica el t_{SM}

Ejercicio 10



Mv (Memoria virtual)

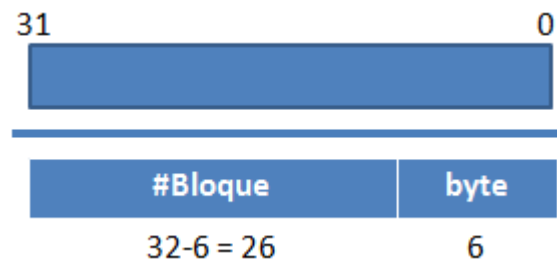
- D_v , 46 bits
- $NTP = 3$ niveles
- TLB
- $P = 8 \text{ KB/pág.} = 2^{13} \text{ B/pág.}$
- $||TP|| = 1 \text{ pág.}$
- 1 pal/PTE_{TP}



Mca (Memoria cache)

- C = 1 MB
- Asoc. sets → 8 líneas/set
- Bq → 64 B/línea = 2^6 B/línea

$$C = \frac{2^{20} \text{ B/cache}}{2^6 \text{ B/línea}} = 2^{14} \text{ línea/cache}$$



Tema 3 – Procesadores ILP

Profesora: M^a Isabel García Clemente

Procesadores con paralelismo a nivel de instrucciones, ILP en sus siglas en inglés (Instruction level Parallelism).

Objetivo

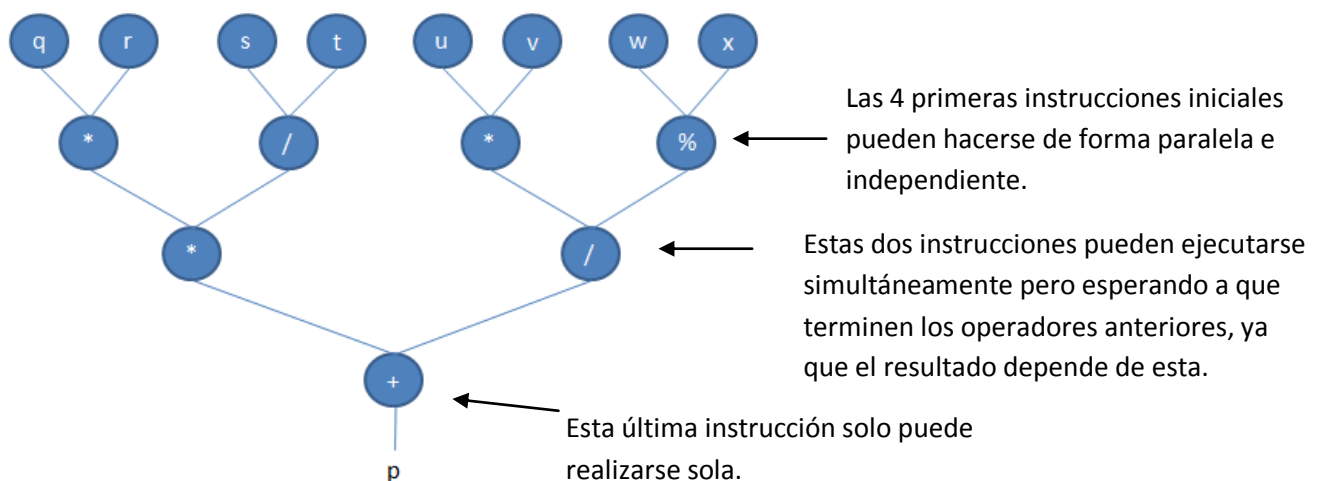
El objetivo fundamental es explorar el paralelismo potencial entre instrucciones de un programa para poder reducir el tiempo total de ejecución de un programa que es el objetivo que se persigue siempre.

Ejemplo

Dada la fórmula:

$$p = ((q \times r) - (s/t)) + ((u \times v) - (w \% x))$$

Dibujamos el árbol de ejecución resultante:



Diferencias de ejecución

Ejecución secuencial

Se ejecuta cada instrucción tras terminar la anterior (terminan todas sus fases).

Pipeline

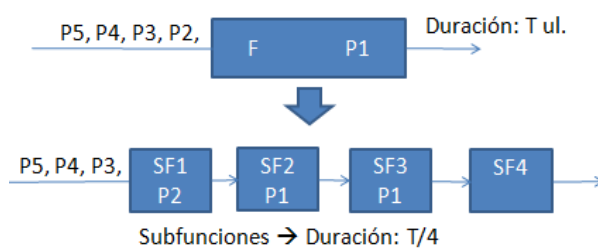
Se ejecuta **cada fase** de una instrucción tras finalizar la misma fase de la anterior instrucción. Al cabo de un tiempo tendremos varias instrucciones ejecutando al mismo tiempo pero siempre en fases diferentes.

Superescalar

Se inician varias instrucciones en el mismo ciclo. Este número depende del grado de escalabilidad del procesador. Obviamente este sistema requiere necesidades que no requiere un procesador escalar, como por ejemplo aumentar el ancho de banda de la memoria (con entrelazado, por ejemplo) y el ancho de banda del bus. También necesita recursos hardware adicionales.

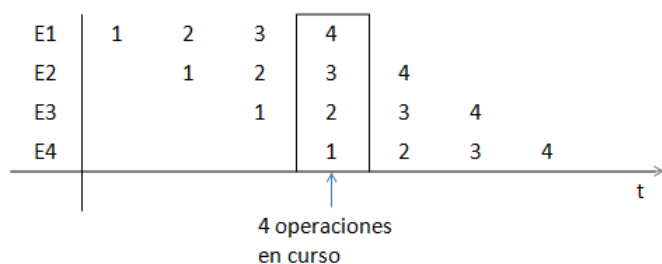
Segmentación (Pipeline)

Si tenemos una función con elementos a procesar de entrada con segmentación lo que queremos es dividir dicha función en subfunciones más pequeñas que se puedan ejecutar simultáneamente.



Donde en cada instante de tiempo cada uno va haciendo una parte del proceso final de forma encadenada. Alcanzamos la máxima productividad cuando todas las etapas duran lo mismo y, en conjunto, tardan lo mismo que la función total.

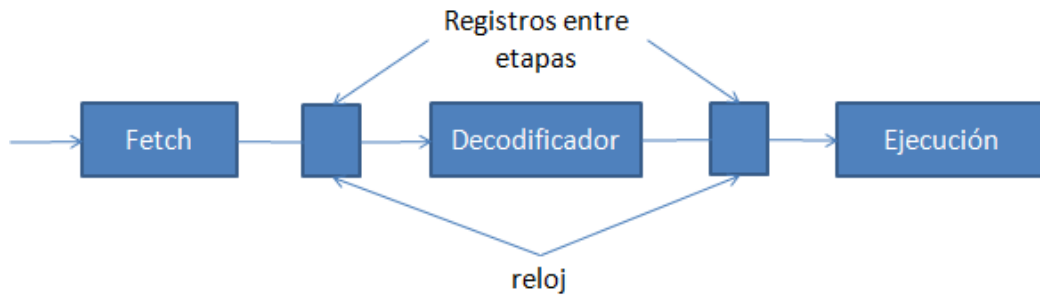
Con este sistema la productividad es cuatro veces mayor (porque hay 4 subfunciones) que en una ejecución secuencial pues en este último sale un resultado cada T unidades de tiempo y en pipeline (en el caso mejor) tenemos un resultado cada $t/4$ unidades de tiempo.



Productividad = throughput

Aceleración = Ganancia = speed-up

Aplicando este concepto al procesamiento de instrucciones podemos dividir cada una de ellas en cada una de las fases de ejecución por las que pasa.



El reloj gobierna el funcionamiento del pipeline y va marcando el ritmo de cada fase. El objetivo es aumentar la productividad, pero debemos tener cuidado con la latencia de una instrucción.

Ciclo de máquina es el tiempo necesario para pasar de una etapa a la siguiente. Controlado por el reloj y está determinado por la etapa más lenta.

Máxima efectividad si todas las etapas tardan el mismo tiempo → Se hace necesaria la memoria cache.

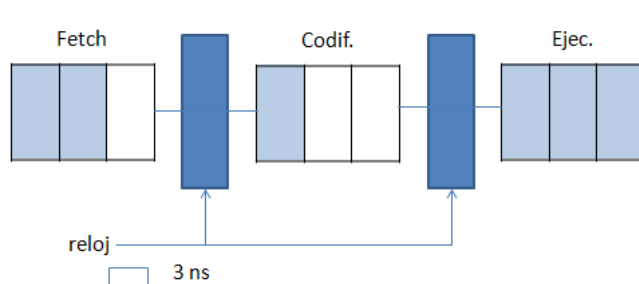
Para codificar el pipeline hay que determinar el tiempo de la fase de Fetch (el más lento en general) que siempre se presupone que es el tiempo de acceso con acierto a la cache L1.

Ejemplo

Fetch: 2 ns Dec: 1 ns Ej(máx): 3 ns

¿T ciclo pipeline?

También sería la duración del ciclo de máquina. Siempre debe ser la velocidad de la fase más lenta. En este caso el tiempo máximo de ejecución es 3 ns.



En el caso peor la fase de ejec. está llena, sin embargo, la fase de fetch y la de codif. deben pasar parte de su tiempo esperando a que pasen los 3 ns.

Latencia

Sin Pipeline

Se suman todos los tiempos → 6 ns

Con Pipeline

Se suma un ciclo de reloj por cada fase, $3 \cdot 3 = 9$ ns. Aumenta la latencia debido a que es una estructura de pipeline desequilibrada.

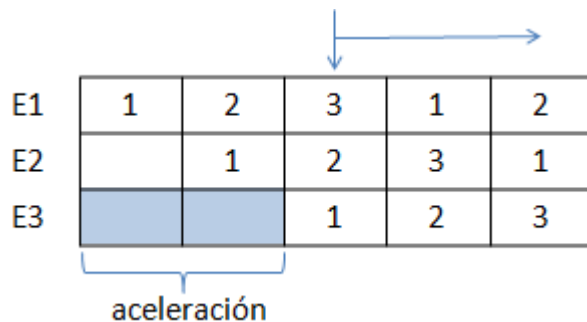
Productividad

Sin Pipeline

1 instrucción por cada 6 ns.

Con Pipeline

1 instrucción por cada 3 ns. La productividad se mide a partir del momento en el que se llena la cadena.



En este caso la aceleración es de 2 cuando lo ideal es que sea igual al número de etapas. No nos encontramos en el caso ideal debido al mal diseño del pipeline.

[FALTAN APUNTES DEL 29 DE ABRIL]

Dependencias

Dependencias RAW

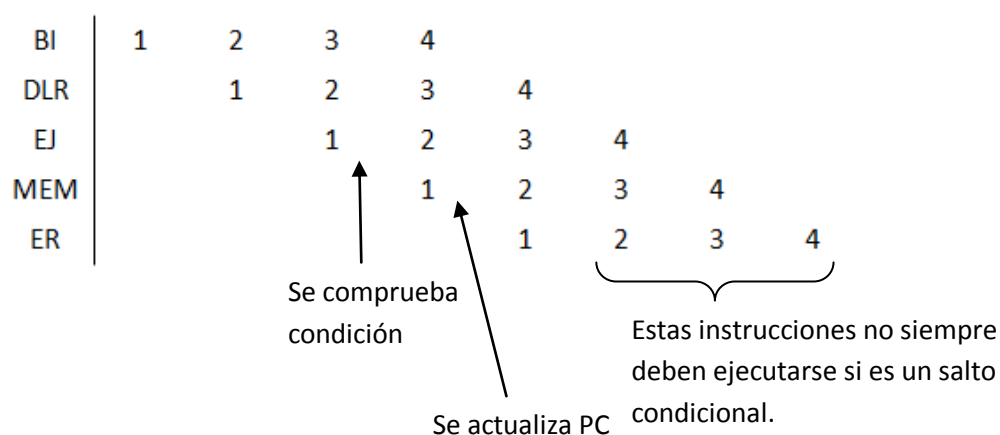
Soluciones de dependencia RAW

Reordenación de código

Si reordenamos el código evitamos las dependencias generadas por el acceso a memoria rellenando esos huecos con otras instrucciones útiles o independientes.

Dependencias de control

Instrucciones de salto, condicional e incondicional.

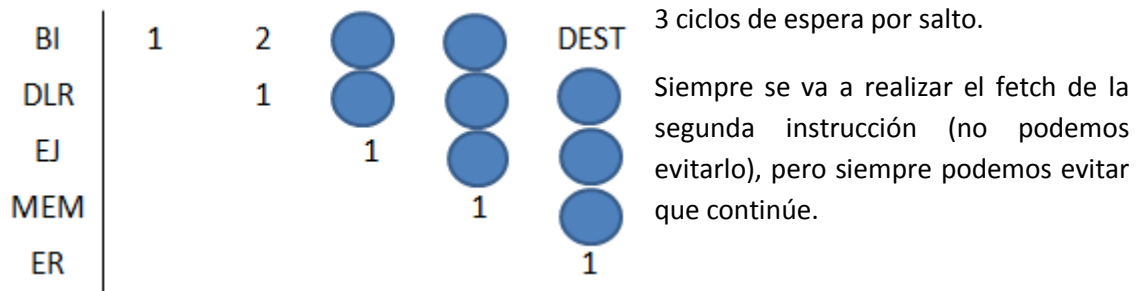


Soluciones

- Inserción de ciclos de espera.
 - Hardware.
 - Software.
- Bifurcación retardada.
- Predicción.

Inserción de ciclos de espera

Solución hardware



Branch delay slots

Añadimos instrucciones de no operación, tantas como Branch delay slots tengamos

```
brcc
NOP
NOP
NOP
```

Podemos observar el CPI en este caso:

$$CPI_{real} = 1 + \underbrace{0,3}_{\text{porcentaje}} \times \underbrace{3}_{\text{Branch delay slot}} = 1,9$$

Ejercicio

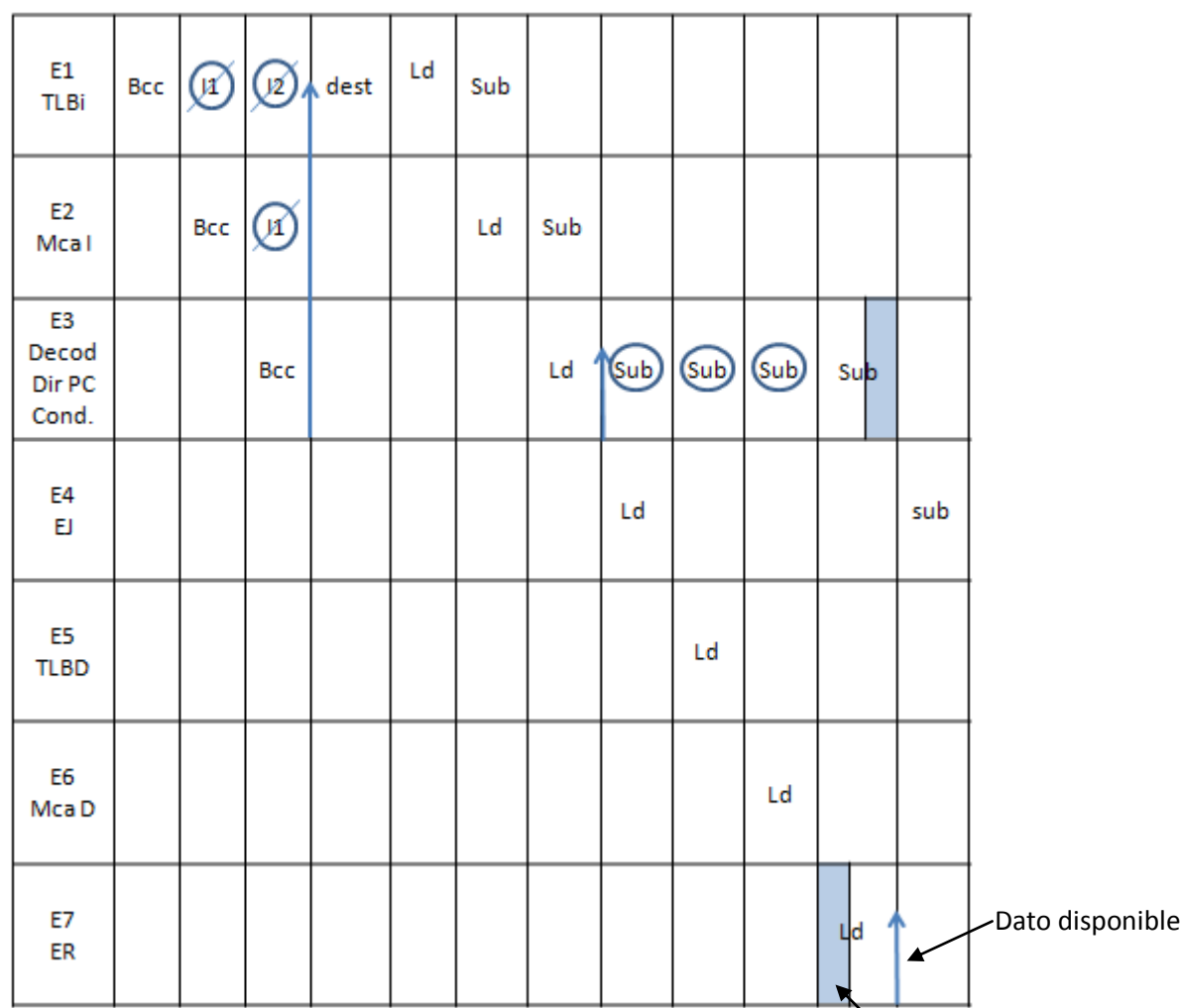
1	LOOP:	ld r1, 0(r11)		Palabra de 32 bits (Instrucción y datos de 1 palabra)
2		ld r2, 0(r12)		
3		sub r3, r1, r2		
4		bgt r3, \$MAYOR	; Si mayor	$C_{acc} \rightarrow 1$ lectura y 1 escritura
5		st r2, 0(r13)		
6		br \$OTRO	; Salto	Pipeline con 7 etapas.
7	MAYOR:	st r1, 0(r13)		
8	OTRO:	add r11, r11, 4		
9		add r12, r12, 4		
10		add r13, r13, 4		
11		sub r4, r4, 1		
12		bnz r4, \$LOOP	; Salto	

```
while(i > 0){
    if(a[i] > b[i]){
        c[i] = a[i];
    }
    else{
        c[i] = b[i];
    }
    i--;
}
```

Dependencias de datos

Existen dependencias de datos entre las instrucciones 2 y 3, entre la 3 y la 4 y entre la 11 y la 12. Para resolverlas añadimos ciclos de espera.

Existen dependencias de datos entre las instrucciones 1 y 3, pero al añadir los ciclos de espera de la dependencia entre la 2 y la 3 también resolvemos esta dependencia.



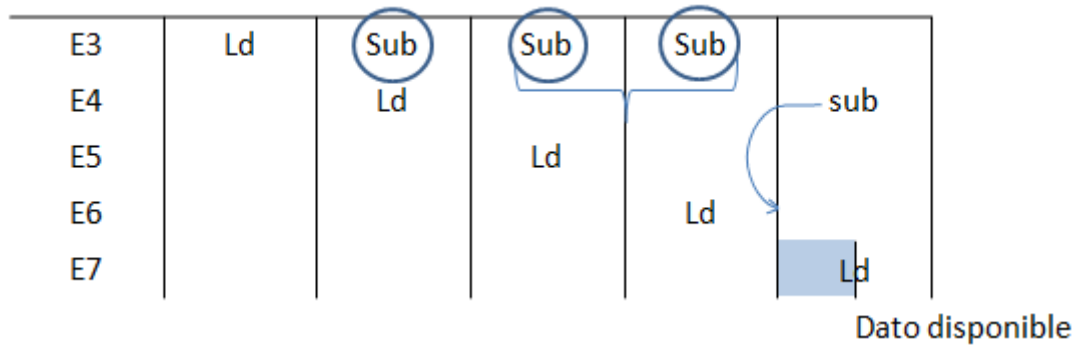
Sin adelantamiento

Se colocan dos ciclos de espera (siempre tantos huecos como ciclos tarde la primera instrucción en actualizar su PC y comprobar la condición). De esta forma resolvemos las dependencias de control.

Se añaden tres ciclos de espera (pues tenemos subciclos) entre la decodificación del dato y su escritura en memoria.

Con adelantamiento

No varían los ciclos de parada debidos a dependencias de control. En la dependencia de datos entre las instrucciones 2 y 3 hacemos un adelantamiento Memoria → ALU que necesita dos ciclos.



En la dependencia de datos entre las instrucciones 3 y 4 (al igual que en la dependencia entre 11 y 12) hacemos un adelantamiento $E4 \rightarrow E3$ que necesita un ciclo.

E1	Ld	Sub	Bgt					
E2		Ld	Sub	Bgt				
E3			Ld	Sub	Bgt	Bgt		
E4				Ld	Sub			
E5					Ld	Sub		
E6						Ld	Sub	
E7							Ld	Sub

$$CPI_{sin\ Ad} = \frac{N \times \left(\overset{n^\circ\ inst.}{\underbrace{4}} + \overset{parones}{\underbrace{8}} + \overset{inst.}{\underbrace{5}} + \overset{parones}{\underbrace{5}} + \overset{No\ se\ cumple}{0,5 \times \left(\overset{ins}{\underbrace{2}} + \overset{par}{\underbrace{2}} \right)} + \overset{Se\ cumple}{\underbrace{0,5 + 1}} \right)}{N \times \left(\underbrace{\frac{4+5}{Inst.fijas}}_{Instrucciones\ empleadas} + \underbrace{\frac{0,5 \times 2}{No\ se\ cumple}} + \underbrace{\frac{0,5 \times 1}{Se\ cumple}} \right)}$$

$$= \frac{24,5}{10,5} = 2,33 \text{ más del doble del ideal (que es 1)}$$

$$CPI_{con\ Ad} = \frac{N \times (4 + 5 + 5 + 3 + 0,5 \times (2 + 2) + 0,5 \times 1)}{N \times (4 + 5 + 0,5 \times 2 + 0,5 \times 1)} = \frac{19,5}{10,5} = 1,86$$

Lo que resulta en una ganancia de:

$$G = \frac{2,33}{1,86} = 1,25 \text{ veces más rápido}$$

No es una ganancia muy grande porque con adelantamiento no ganamos ciclos de espera por dependencia de control.

Reordenación del código

Al introducir ciclos de espera estamos dejando espacio vacío que provocarían una parada en el proceso. Para evitar esto y para que la CPU siga funcionando podemos reordenar el código para rellenar esos huecos libres con instrucciones independientes. En nuestro caso introducimos las instrucciones 8 y 9 entre las instrucciones 2 y 3 y la instrucción 11 entre la 3 y la 4. Suponemos adelantamiento.

$$CPI_{reord} = \frac{N \times (4 + 2 + 5 + 2 + 0,5 \times (2 + 2) + 0,5 \times 1)}{N \times (4 + 5 + 0,5 \times 2 + 0,5 \times 1)} = \frac{15,5}{10,5} = 1,47$$

Con una ganancia:

$$G = \frac{1,86}{1,47} = 1,26$$

Pero ya no tenemos dependencias de datos.

Ejercicio de examen

Procesador con varias UF:

- div → 8 ciclos
- enteros → 1 ciclo
- mul → 3 ciclos

y 4 etapas pipe:

- Fetch
- DLR
- EJ
- ER

Dado el siguiente fragmento de código:

```
1    div r2, r5, r8
2    sub r9, r2, r7
3    add r5, r14, r6
4    mul r2, r9, r5
5    mul r10, r11, r12
```

Calcular:

a) Posibles dependencias de datos.

b) Evolución de las instrucciones en pipeline suponiendo:

- Sin planificación dinámica.

- Introduce ciclos de datos para resolver dependencias de datos.

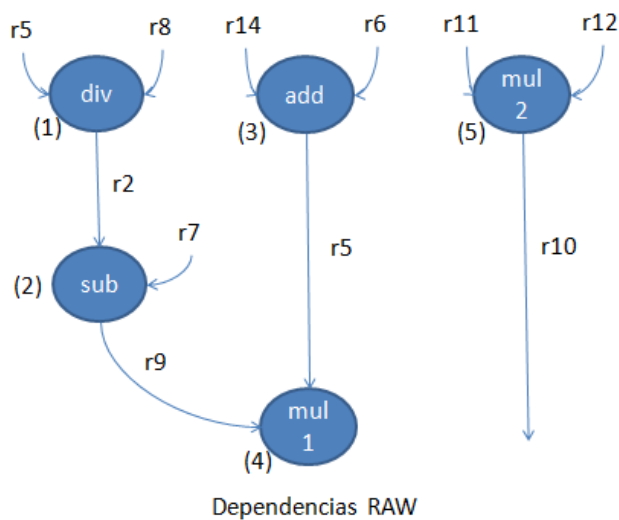
c) Diferencias entre este comportamiento y procesadores con planificación dinámica.

Solución

Apartado a

Posibles dependencias:

- Entre 1 y 2 por r2 (RAW).
- Entre 3 y 4 por r5 (RAW).
- Entre 1 y 3 debido a lo lenta que la UF div (WAR).
- Entre 1 y 4 (WAW).
- Entre 1 y 4 (RAW).



Apartado b

Sin planificación dinámica

Las dependencias RAW se solucionan mediante la introducción de ciclos de espera. El resto no dan problemas en un pipeline sin planificación dinámica.

E1	1	2	3								3	4	5		5					
E2		1	2*								2	3	4**		4	5				
E3			1	1	1	1	1	1	1	1		2	3		4	4	5	4	5	
E4											1		2	3					4	5

*2 se para por dependencias de datos.

** 4 se para por dependencias de datos.

Escribe al finalizar el ciclo (por no tener división de ciclos).

Efecto del Pipeline en el multiplicador.

Ciclos de datos para resolver dependencias de datos.

E1	1	2	3	4	5														
E2		1	2*	3	4**	5													
E3			1	1	1	3	1	1	5	1	5	1	5	1	**2		4	4	4
E4							3						5	1		2			4

*2 se queda esperando es una estación de reserva y, si no existe dependencia entre sí, la siguiente instrucción le adelanta.

**4 se queda esperando en una estación de reserva.

*** La resta se dispara automáticamente y pasa a la fase de ejecución.

No se dan las dependencias WAR porque las instrucciones toman los datos en la fase 2 y actualizan el estado en la fase 4.

La dependencia WAW es una dependencia falsa ya que, en este caso, la instrucción 4 tiene dependencia de otro tipo que ya se resuelve, por tanto la dependencia WAW no da problemas.

Sin embargo, si quisiéramos solucionarlas en general tendríamos que renombrar registros de forma estática:

```

1    div r2, r5, r8
2    sub r9, r2, r7
3    add r20, r14, r6      ; r5 <-> r20
4    mul r21, r9, r20      ; r2 <-> r21
5    mul r10, r11, r2

```

Apartado c

Diferencias:

Orden de ejecución:

1, 2, 3, 4, 5 vs 1, 3, 5, 2, 4

Orden de terminación (actualizar estado):

1, 2, 3, 4, 5 vs 3, 5, 1, 2, 4

Velocidad:

21 ciclos vs 17 ciclos

Tema 4 - Multiprocesadores

Profesor: Antonio García Dopico

Ejecución paralela

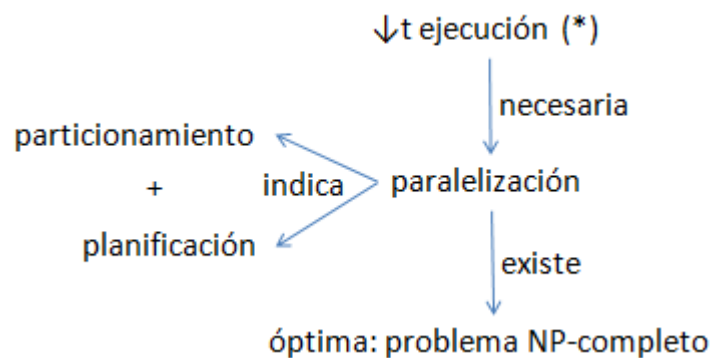
¿Es necesaria? Sí, porque:

- Demanda de cómputo. Aplicaciones de modelado que necesitan mucha memoria y, por tanto, más lentas.
- Logros tecnológicos. Silicio,...
- Logros en arquitectura. Superescalares, VLIW,...
- Aspectos económicos. Duplicar potencia sin duplicar precio.

Beneficios del paralelismo

¿Merece la pena? Sí, porque:

- Forma fácil: multiproceso \uparrow throughput. Acabamos antes ya que en cada core ejecutamos un proceso independiente.
- Forma (muy) difícil: ¡El gran reto!



Puede paralelizarse automáticamente mediante el compilador, el sistema de ejecución o el SO y también manualmente (con mayor o menor trabajo del programado): Occam, Ada, HPF, PC++, PVM, MPI, **OpenMP**,...

Paralelismo a nivel de bit \subset ILP \subset Vectorial \subset Multicore \subset Nodos multiprocesador \subset Cluster

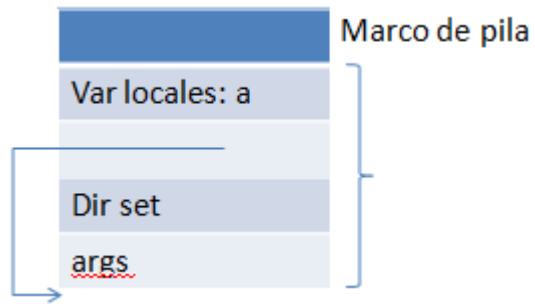
Cada tecnología moderna lo que trata es de mejorar la anterior.

Ejercicios

¿El problema de la coherencia de caches tiene relación con los registros de la CPU? ¿Cómo se garantiza la coherencia de los datos en registro? ¿Solución Hw o Sw?

Solución software, depende del compilador y del programador. Para solucionar este problema utilizamos `volatile` que indica al compilador que es un dato peligroso que puede ser modificado desde otro lugar. Por ejemplo:

```
f(i,j){
    volatile int a;
    ...
    a = b + c;
    ...
    return(c);
}
```



Normalmente la variable `a` se copia en un registro para trabajar con ella más rápidamente, pero si se hace así alguien puede modificar el registro y puede causar problemas. Con `volatile` indicamos al compilador que no haga esa copia en cache y mantenga la variable en el marco de pila.



Apuntes de Arquitectura de Computadores by [Pau Arlandis Martínez](#) is licensed under a [Creative Commons Reconocimiento 3.0 Unported License](#).